



Node.js

来一打 C++ 扩展

死月 著

电子工业出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

Node.js 作为近几年新兴的一种编程运行时，托 V8 引擎的福，在作为后端服务时有比较高的运行效率，在很多场景下对于我们的日常开发足够用了。不过，它还为开发者开了一个使用 C++ 开发 Node.js 原生扩展的口子，让开发者进行项目开发时有了更多的选择。

本书以 Chrome V8 的知识作为基础，配合 GYP 的一些内容，将教会大家如何使用 Node.js 提供的一些 API 来编写其 C++ 的原生扩展。此外，在后续的进阶章节中，还会介绍原生抽象 NAN 以及与异步相关的 libuv 知识，最后辅以几个实例来加深理解。不过，在学习本书内容之前，希望读者已经具备了初步的 Node.js 以及 C++ 基础。

阅读本书，相当于同时学习 Chrome V8 开发、libuv 开发以及 Node.js 的原生 C++ 扩展开发知识，非常值得！

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目 (CIP) 数据

Node.js: 来一打 C++ 扩展 / 死月著. —北京: 电子工业出版社, 2018.6

ISBN 978-7-121-33642-3

I . ① N… II . ① 死… III . ① C++ 语言—程序设计 IV . ① TP312.8

中国版本图书馆 CIP 数据核字 (2018) 第 022884 号

策划编辑: 刘 蛟

责任编辑: 李云静

印 刷: 三河市华成印务有限公司

装 订: 三河市华成印务有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编: 100036

开 本: 787×980 1/16 印张: 30 字数: 655 千字

版 次: 2018 年 6 月第 1 版

印 次: 2018 年 6 月第 1 次印刷

定 价: 109.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

好评袭来

This book contains absolutely everything you need to know about how all the pieces of Node.js' C++ code work and interact, explaining the necessary concepts without needing prior knowledge about the internals of V8, libuv or other pieces of Node.js. It shows well how Node.js' own built-in modules are constructed using the APIs provided by V8, so that they are usable from JavaScript, and how you can create the same kind of modules from scratch.

After having read this book, you will be able to write a production-quality, future-proof C++ extension for Node.js if you need to do that, or maybe even make changes Node.js itself if you're interested in that!

本书包含了所有你需要了解的有关 Node.js C++ 代码是如何运行和交互的知识，解释了一些你不需要知道 V8 的内部机制就能理解的必要概念，并介绍了 libuv 以及其他一些内容的方方面面。本书还展示了 Node.js 的内置模块是如何使用 V8 的 API 进行构建并可在 JavaScript 层面使用的——并且你也能用这种方法从头开始创建相同类型的模块。

读完本书，你将学到如何写出产品级质量的、面向未来的 Node.js C++ 扩展。感兴趣的话，你甚至可以对 Node.js 自身进行修改！

——安娜·亨宁森（Anna Henningsen, addaleax）
Node.js 技术指导委员会成员（Node.js TSC）

Node.js 不是第一个将 JavaScript 带入服务端领域的技术，然而它却成为史上最热门、最有影响力的工具之一。究其原因，其一，在于 Node.js 适逢后端高并发潮流，巧妙结合 Reactor 模型和 JavaScript 所擅长的回调风格，大大降低了开发高并发服务器应用的成本；

IV Node.js: 来一打 C++ 扩展

其二，在于恰逢浏览器大战，前端技术突飞猛进，急需一套适合 JavaScript 和前端工程师的生态和工具链，Node.js 刚好成为前端 JavaScript 最易上手掌握的命令行环境。在 Node.js 发展得如此火热之际，Node.js 的开发体验在不断提升，上手门槛也在不断降低。

然而，如果大家真正想突破自己并成为个中高手，无论是后端程序员希望在服务端及架构方面有所建树，还是前端程序员想跨越边界，都应该去了解 Node.js 的底层机制，去学习写一些 Node.js 的扩展。从 Node.js 的内在机制，我们可以学到更多有关计算机体系的知识，如内存管理、多线程编程等，真正向一个架构师、一个大牛迈进。

本书在这些方面提供了一个非常系统的指南。死月通过精彩的内容告诉大家：底层的知识并不枯燥，用 C++ 写一个扩展很有意思也很简单。作为 Node.js 工程师 / 爱好者的你，值得拥有本书。

——曹力（ShiningRay）

酷链科技 CEO，暴走漫画前 CTO，糗事百科前联合创始人，
高级 Node.js 技术专家，《JavaScript 高级程序设计》译者

Native module is one of the most underappreciated features of Node.js. But even in the age of asm.js and WebAssembly, it is an irreplaceable part of the Node.js ecosystem due to its versatility and performance. XadillaX's book provides a refreshing introduction (or reintroduction), and is a must-read for all low-level Node.js engineers.

原生模块是 Node.js 中最被低估的功能之一。因为自身的性能和多样性，使其即使是在 asm.js 和 WebAssembly 时代，仍旧能作为 Node.js 生态系统中不可替代的部分存在。死月的书对其进行了一个令人耳目一新的介绍，它是所有底层（Low-Level）Node.js 工程师的必读之物。

——顾天骢（Timothy Gu）

pug、ejs 前 Maintainer，Node.js Core Collaborator 之一

本书全面讲解了 V8、libuv 的原理并且手把手教你编写一打 Node.js 的 C++ 扩展，它是目前市面上相关领域非常稀缺的技术书籍。如果你想更深入地了解 Node.js 的实现原理，除了熟读内置 API 文档之外，阅读本书也会是一个很好的选择。

——雷宗民（老雷）

《Node.js 实战》作者之一

这是一本角度“刁钻”的 Node.js 相关书籍，其与市面上大多数 Node.js 书籍的定位不同。本书借为 Node.js 开发 C++ 扩展做基石，顺带介绍了 Chrome V8 和 libuv 的内容，填补了市场上这一类书籍的空白，值得一读。

——李启雷博士
趣链科技 CTO

无论是基础部分的 V8 练习，还是使用 Node.js 经典的 Addon 开发、用 NAN 来改写，或是 libuv 里的 WatchDog 案例、EFSW 的封装，死月一直把实战贯穿在整本书之中。甚至在第 8 章里他还特意剖析了两个 C++ 模块，把之前讲解的基础知识部分综合起来，以便让读者可以边学边练。

在如今追求大而全的时代，这本《Node.js：来一打 C++ 扩展》单纯地讲 Node.js 的某一个方面，而且讲得特别棒，真的很难得。

——刘琥（响马）
西祠胡同创始人，fibjs 作者

当你掌握了 Node.js 的上层使用，下一步进阶的方向就是研究 Node.js 的底层原理。本书为学习 Node.js 的实现机制打开了一扇门。书中介绍的上下文（Context）、句柄（Handle）、句柄作用域（Handle Scope）等概念直接来自源码，对于阅读 Node.js 及 V8 的源码具有极高的参考价值。

——潘旻琦（pmq20）
Node.js 技术专家，Node.js Collaborator 之一，RubyConf 讲师之一

国内 Node.js 偏向于原理的书目前只有朴灵的《深入浅出 Node.js》一本。至今 4 年过去了，Node.js 的版本已经从 v0.10 发展到 v9，中间几乎没有这样系统、有深度的书籍。

很高兴死月的新书弥补了这一遗憾。本书以 C++ 为主线，涵盖 Node.js 最核心的 libuv 和 V8，对理解 Node.js 原理有极大的好处。当然最大的好处在于，使用 C++ 编写 Node.js Addon 可以让 Node.js 有更广阔的应用空间。我们都知道 Node.js 擅长的是 I/O 密集型任务，对于 CPU 密集型运算这是极好的弥补。

特别推荐大家阅读此书，在 Node.js 应用极其广泛的今天，使用 C++ 编写 Node.js Addon 是更出彩的部分，你值得拥有。

——桑世龙（i5ting、狼叔）
StuQ 明星讲师，Node.js 技术布道者，《更了不起的 Node.js》作者

VI Node.js: 来一打 C++ 扩展

死月对 Node.js 底层机制有非常深入的了解。阅读本书，除了学习 C++ 扩展开发外，还会跟随死月了解 V8、libuv，相信大家对于 Node.js 的理解会更上一层楼。

——孙信宇（芋头）

大搜车无线架构团队负责人，前端乱炖站长

C++ 扩展其实是从外在，用 C++ 的角度去观察 Node.js 内在的形式。因为 Node.js 整个系统自身几乎就是构建在 C/C++ 之上的，所以 C++ 扩展只是在 Node.js 内部被称为内置模块（built-in module），在第三方角度（user-land）则被称为 Addon，它们在本质上其实没有区别。死月凭借他在 C/C++ 方面的深厚积累，选择从 C++ 扩展作为突破口，带大家领略 Node.js 底层的风光。在本书中，你能看到真正发挥巨大价值的 V8、libuv 亦是精彩纷呈。

死月将 C++ 扩展写得这么透彻，我是服的。

——田永强（朴灵）

高级 Node.js 技术专家，《深入浅出 Node.js》作者

开发 C++ 扩展，可以扩充 Node.js 平台的本地 API，扩充 Node.js 应用的能力。本书详细介绍了包括 libuv、V8 在内的各种必要知识，是该领域比较难得的好书。对 C++ 开发者来说，本书既可以作为入门指引，又可以作为日常开发的有益参考。

——王文睿博士（Roger Wang）

node-webrtc 和 NW.js 项目创始人和维护者，英特尔软件架构师

我至今仍然清晰记得，自己手写的第一个 Node.js C++ 扩展模块在 Node.js 0.6.9 跑通的那种愉悦感。随着应用升级到 Node.js 0.8，依赖的 C++ 扩展模块无法安装编译成功，最后发现是 V8 的 API 变化导致不兼容，从此我对 C++ 扩展模块产生了抗拒心理。如今看到《Node.js：来一打 C++ 扩展》从实现原理到 V8 基础概念的一系列介绍，让我重新对 C++ 扩展模块产生了兴趣。参考本书中的实战例子，并在 NAN 的辅助下，编写一个跨 Node.js 版本的 C++ 扩展已经不是什么困难的事情了。通过最后一章，读者可以了解到 Node.js 官方的 N-API 计划，让 C++ 扩展不仅仅能跨版本复用，还能跨操作系统（平台）复用。

——袁锋（fengmk2）

Node.js 技术专家

序一

1995 年 Brendan Eich 花了 10 天时间开发出了一门脚本语言，用来弥补 Java Applet 的不足，随后 Marc Andreessen 给它起名为 Mocha。其最初的定位是，Java 用于大型专业级开发，而 Mocha 则是给测试脚本编写人员、业余爱好者、设计师使用的。

1995 年 5 月，Mocha 被集成到了 Netscape 浏览器中，其不久后改名为 LiveScript，当年年底网景公司和 Sun 公司达成协议并获得了 Java 商标的使用权，其正式更名为 JavaScript。

有人说 Sun 公司的介入限制了 Brendan Eich 的手脚。JavaScript 除了某些语法和 Java 类似以外，骨子里却是完全不一样的东西。

也有人说正式改名为 JavaScript 才使得这门语言成为浏览器执行的唯一语言。

时至今日 JavaScript 已经不仅仅局限于为网页做特效了，而真正发展成为一门全功能的编程语言：

- 2008 年 Chrome 发布、V8 发布；
- 2009 年 Node.js 发布；
- 2010 年 NPM 发布；
- 2014 年 12 月，多位核心 Node.js 开发者不满于 Joyent 对 Node.js 的管理制度，创建了 io.js；
- 2015 年初 Node.js 基金会成立；
- 2015 年 9 月 Node.js 4.0 发布，Node.js 和 io.js 正式合并。

Node.js 4.0 版引入了 ES6 的语言特性和“长期支持版本”的发布周期。

如今 Node.js 社区已经成为最活跃的编程社区之一，而从 NPM 的包数量来看，其已经超越了 Java 的 Maven、Ruby 的 gem、PHP 的 composer。

VIII Node.js: 来一打 C++ 扩展

但是 Node.js 仍有很多不足之处，Node.js 的使用者绝大部分仅仅把 Node.js 作为前端开发的辅助工具。大家把 Node.js 作为后端主力开发平台使用时，遇到 CPU 密集的场景时又不得不借助 Java 或者 Go。虽然 V8 引擎一直致力于让 JavaScript 运行得更快，但是和 Java、C++ 相比，还有不小的性能差距。

虽然关于 JavaScript 的书已经汗牛充栋，但是有关 Node.js 原理的书却屈指可数。而目前真正能够深入介绍原理的，国内的图书中也只有朴灵的《深入浅出 Node.js》了，但如今四五年过去了依然没有等到该书的第 2 版，而死月的这本书却可以弥补这一方面的不足。

所有的编程语言底层都会回归 C/C++，Node.js 的底层依赖库 V8 使用 C++ 开发，libuv 则使用 C 语言。而使用 C++ 开发 Node.js 扩展将直接把擅长 CPU 的 C++ 和擅长 I/O 的 Node.js 结合在了一起，弥补了 JavaScript 在计算密集型应用方面的不足。

我从 2015 年开始研究 V8，认识死月的时间则更早。死月不仅仅精通 C++，他也是国内的 Node.js 布道师之一。从我认识他起，他就一直在使用 Node.js。如果你想深入了解 Node.js 的原理，或者想打开 Node.js 另一个世界的大门，这本《Node.js: 来一打 C++ 扩展》值得你精读。

——迷渡（justjavac），Flarum 中文社区创始人，国内知名前端技术专家

2018 年 3 月 22 日于天津

序二

我跟死月相识于 GitHub，那时我们经常会向 Node.js 贡献一些代码，彼此也会在微信上讨论一些技术问题。当我听说死月在写一本关于 Node.js C++ 扩展相关的图书时，激动得几乎要从床上蹦起来。因为我深知一个对 Node.js 与 V8 引擎都如此了解之人，愿意将他所知所想分享出来，这将是给予社区的一份大礼。

从我个人的角度来看，这本书非常适合这类开发者：他们对于 Node.js 的使用已经了然于胸，但却苦于没有底层开发经验，对整个 V8 虚拟机也一知半解。这时，他们可以从第 3 章开始读起。本书用了很长的篇幅介绍 JavaScript 代码究竟在虚拟机里是怎么运行的，它们又都分别对应着哪一类数据结构等。因为作者深知，只有把这些基础理解透了，则无论是开发 C++ 扩展，还是写纯 JavaScript 代码，大家都能更得心应手。

本书像是在述说着 Node.js 在 C++ 扩展这一课题中曲折而又有趣的历史进程。首先从最原始的 V8 API 时代开始。对于每个原始时代，开发者最痛苦的莫过于解决各种版本的兼容问题。之后迎来的是 NAN 时代。它解决了原始时代的接口抽象问题，接口也更丰富多样，异步接口也封装在内。最后，是还在路上的 N-API。它与 NAN 一脉相承，拥有更官方的支持和更友好的接口。

另外，我们通常在写一个 C++ 扩展时，多数情况下会跟异步打交道，这其中包含着如何非阻塞地调用底层接口，如何将异步的结果返回到 JavaScript 的回调函数中，以及如何正确地在异步封装中释放你的资源。对这些内容特别感兴趣的读者，可以打开第 6 章一睹为快。

Node.js 已快走完它的第一个 10 年，尽管被人诟病于其回调地狱、虚假繁荣、超高并发场景下的不适应性，以及低端设备上的内存等问题，但这仍旧无法阻止它前进的步伐。然而对于我们 Node.js 工程师来说，除了掌握好这门语言之外，学习如何写 C++ 扩展、了解它如

X Node.js: 来一打 C++ 扩展

何运转将是我们下一阶段的重要功课。相信《Node.js: 来一打 C++ 扩展》将会成为常伴大家左右的另一本《代码大全》。

——刘亚中（Yorkie），Rokid 系统工程师，tensorflow-nodejs 作者

2018 年 3 月 22 日于杭州

前言

写这本书是我在 2016 年底许下的愿望，希望在 2018 年初完成一本技术专著。

我于 2012 年加入 Node.js 开发的大军，现在也有幸成为 Node.js 这个项目的 Core Collaborator 之一。所以，我的意向就是为大家呈现一本 Node.js 领域相关的书。但是现在市面上相关的书籍其实有很多了，我再写一本日常开发类的图书就显得有些多余。反而是在 Node.js 的 C++ 扩展开发方面，无论是在国内还是在国外，都是一块死角。就目前而言，国外市场我也只看到过一本电子书，并没有纸质图书出版，国内就更没有了。

Node.js 作为近几年新兴的一种编程运行时，托 Chrome V8 引擎的福，在作为后端服务时有比较高的运行效率，在很多场景下对于我们的日常开发已经足够用了。不过，它也跟“PHP 提供了 C 语言开发其原生扩展的方式”类似，为开发者开了一个使用 C++ 开发 Node.js 原生扩展的口子，让开发者进行项目开发时有了更多的选择。

实际上，在 Node.js 的生态圈中，就有很多使用 C++ 完成的包。如最近比较火的深度学习 TensorFlow，其 Node.js 版本的封装就是基于官方的 C++ 源码完成的。我自己就是在日常开发中有一些相应的需求，使用纯粹的 Node.js 来开发可能会使开发成本有点大，或者基本上做不到，又或者有性能上的要求。这时，我就会选择使用 C++ 来实现它的一个扩展。在我写了一段时间 C++ 扩展之后，想到可能在社区中有很多像我一样的人，苦于 Node.js 在底层操作时的一些局限性，如果他们也加入 C++ 原生扩展开发阵营的话，兴许要再踩一遍我以前踩过的“坑”，找我之前找过的资料。因此，我就想把自己一路走来的经验分享给大家，让更多的人顺利地加入 Node.js 的 C++ 原生扩展开发的大军中。

我的 Node.js 之路

我个人从小学开始接触静态网页的开发，直到高中开始参加信息学奥林匹克竞赛（Olympiad in Informatics, OI），才算正式踏入了编程之路。

在大学的时候我仍旧坚持参加大学生程序设计竞赛（ACM International Collegiate Programming Contest, ICPC），并且一直使用 C++ 和 PHP 进行开发。也是那时我打下了 C++ 基础，这样才有机会现在完成这本书的写作工作。

我接触 Node.js 其实并没有国内一些早期的布道者们早，相反还是有点迟的。在 2012 年底，我决心学习 Node.js，从而完成自己的一个创业项目。我当时的学习方法特别简单，买了一本 BYVoid 的《Node.js 开发指南》，就算正式踏入了 Node.js 领域。

在熟悉了 Node.js 之后，我开始为 Node.js 生态圈造轮子，如 Toshihiko¹、Thmclrx²、Huaming³、mcnbt⁴ 等。其实我个人认为，造轮子与写业务的一个不同点在于，造轮子可能会更容易遇到语言或是 Node.js 运行时本身的“坑”。所以，这就促使我去深究 Node.js 的文档，甚至源码。托我之前习得的 C++ 基础的福，在阅读 Node.js 源码时并不觉得特别艰难。

我在老东家花瓣网的时候，就已经初步开始了 Node.js 的 C++ 扩展开发。

后来我去了上一家就职的公司——大搜车，负责公司 Node.js 团队的建设。当时我就开始更深入地挖掘 Node.js 的一些内容了。甚至在 2017 年年中的时候，我通过给 Node.js 贡献源码，成为 Node.js Core Collaborator 之一。我在为 Node.js 贡献源码的时候，也为本书第 3 章和第 6 章的写作打下了基础。

原生扩展的一些示例

在 Node.js 早期的版本中，运行子进程是纯异步的，并没有像现在一样的各种 `spawnSync()` 等函数。我当时写了一个命令行工具，在其所用到的一个帮助类中实现一个参数校验的函数必须同步返回一个布尔类型的值；然而我在这个校验函数中所需要做的事情就是判断当前系统的 Git 版本。也就是说，我要通过子进程启动 `$ git -v`，并得到它的结果看看版本是不是符合要求。当时 Node.js 中运行子进程是异步的，达不到我的要求，所

1 一个极简、自带黑盒缓存层的 ORM，详见 <https://github.com/XadillaX/Toshihiko>。

2 一个图片主题色提取的包，详见 <https://github.com/XadillaX/thmclrx>。

3 随机起名神器，详见 <https://github.com/boogeedoo/hua>。

4 Minecraft 游戏的 *.nbt 文件解析库，详见 <https://github.com/BoogeeDoo/mcnbt>。

以我自己使用 C++ 封装了一个原生扩展，使其能在 Node.js 的事件循环中同步开启子进程并在其结束后获得它的终端输出——虽然 Node.js 天生异步，但是在我自己的一个命令行工具中用同步形式执行这些内容也是没有问题的。

再比如，在大搜车的时候，项目用了阿里云的消息队列服务，而这款产品当时只有闭源的 Java、C++ 等 SDK，而 C++ 的 SDK 就只提供了几个动态链接库和一堆头文件，我们使用 Node.js 的开发者就完全没法使用其服务。如果一定要用 Node.js 进行开发，一个成本比较高的做法就是自己去逆向分析及研究消息队列服务的各种网络包的结构，自己解析，然后用 Node.js 实现一个同样功能的库。然而这个方法基本不可行——尤其是在我们的项目高速迭代的时候。那么另一个办法就是基于其闭源的 C++ SDK，使用本书中的各种开发方式，写一份 Node.js 的 C++ 扩展。这样就能把它们 C++ SDK 集成到我们的 Node.js 项目中了。这是一个非常好的降低开发成本的方法。

当年我还在花瓣网的时候，有一个需求是提取一张图片的主题色。我当时翻阅了不少论文，最终采用了一种八叉树加最小差值法的结合体¹来完成这个需求。在数据结构和整型数字处理方面，我个人认为 C++ 的开发效率和执行效率比 Node.js 要高，于是我自然而然地就使用了 C++ 把核心算法部分完成了（现在我甚至使用 C 语言又重构了一套，开源在 GitHub 上面²）。然后为了将其集成到我们的 Node.js 任务调度系统中，我又将其封装成了一个 Node.js 的 C++ 的扩展。这样一来，主题色提取的任务就欢快地运行了——它也被开源在我的 GitHub 上面，就是前面提到过的 ThmclrX。而且借这个包的“东风”，我的硕士毕业论文写的就是这么一套主题色提取的任务系统相关内容。

类似的案例还有很多。如计算字符串哈希值等，由于用 JavaScript 重写代码的时候，在整数的各种操作上会有很多“坑”，因此拿 C++ 源码封装一下就非常简单的。甚至谷歌推出的 CityHash 这个算法只有一份冗长的 C++ 源码，使用 JavaScript 重写的话将会是一个比较庞大的工作量；再比如解析 MP4 文件的时长，我个人不是多媒体相关领域的开发者，所以并不擅长。于是我弄了一份 C++ 的源码，懒得转换——嘿，用 C++ 扩展一包，直接就发布了；还有同步获取 HTTP API 的内容，写一个能继承的类似于 ECMAScript 6 中 Proxy 特性的拦截器；等等。

1 八叉树提取法加最小差值法的混合算法，详见我的博客《图片主题色提取算法小结》一文：<https://xcoder.in/2014/09/17/theme-color-extract/>。

2 C 语言的主题色提取库，“圣·白莲”，详见 <https://github.com/XadillaX/byakuren>。

本书面向的读者

在阅读本书前，我希望你对 Node.js 比较熟悉，并且对于 C++ 这门语言至少要有一个初步的认识。当然，如果你的 C++ 基础并不是很好的话，也不要怕，可以多读几遍本节最后的一段话。

本书不仅仅讲实践，我还花了不少篇幅来讲解它的前驱知识，如 Chrome V8 引擎开发的一些基本概念，如句柄、句柄作用域等，以及各种 API 的初步介绍。另外，书中还介绍了 libuv 层面的内容，尤其是在异步方面，像 libuv 中的线程、同步原语，以及如何在 Node.js 的主时间循环中与你自身写的线程进行跨线程通信等。这么一算，Chrome V8、libuv，加上 Node.js 的 C++ 扩展开发，你相当于一下子买了 3 本书，是不是觉得很超值？**也就是说，你阅读本书的目的不一定是想要开发 Node.js 的 C++ 扩展；如果你想学习 Chrome V8，或者想学习 libuv，也可以参考本书。**

本书的最后还简单展望了一下 Node.js 8.0 之后出现的一个新特性，就是新一代 Node.js C++ 原生扩展接口 N-API。不过由于 N-API 还处于试验阶段，各种接口还不是很稳定，在未来随时会变，因此本书中并没有详细地介绍 N-API，而只是简单讲解了它的思想，让大家在心中有一个思想准备。这样，哪一天 N-API 正式发布了，读者就可以比较快地上手了。不过，不要忘本，哪怕 N-API 真的出来了，我还是希望大家多了解一下底层的基础，比如像 Chrome V8、libuv 以及 Node.js 源码相关的内容。因为学习了这些基础知识，对大家肯定没有坏处（甚至对于 Node.js，大家说不定会有一个新的认识）。

最后，奉上我在一次技术直播中说过的一句话：“当我们在学习 Node.js 的时候，我们其实就是在学编程。语言只是最表象的东西，思想才是核心内容。”如果还有部分读者由于本书需要有 C++ 基础望而却步的话，多读几遍我刚才说的话，然后鼓起勇气入“坑”吧。

本书的结构

本书共分为 9 章。其中前两章描述了一些基础的前驱理论知识；第 3 章到第 6 章讲的是 Node.js 的 C++ 扩展开发中用到的各种知识，并辅以简单的样例；第 7 章和第 8 章为实战章节，根据现实需求来完成相应的 Node.js C++ 扩展；第 9 章为对未来的 N-API 的一个展望。

第 1 章讲述了我们在学习本书内容之前所需要了解的基础，如 Node.js 的模块机制与包机制，以及 Node.js 都是由什么三方依赖构成的。其中就提到了很重要的 Chrome V8 和

libuv。本章的最后还讲述了要进行 Node.js 的 C++ 扩展开发所需要做的准备工作，包括但不限于编辑器的挑选、开发环境的搭建等。

第 2 章主要讲述了什么是 Node.js 的 C++ 扩展，它的本质是什么，并且什么情况下需要使用 C++ 扩展，以及阐述了为什么在这些情况下要使用 C++ 扩展。

第 3 章介绍了谷歌的 Chrome V8 引擎，从它与 Node.js 的关系讲到它的一些基本概念，例如 V8 的内存机制、基本对象等。在后续的章节中将开始介绍 Chrome V8 的各种类及其概念，以及它们的用法，如句柄、句柄作用域、模板和各种常用的数据类型等。

第 4 章相当于各种编程语言书籍中的“Hello World”，向读者介绍了 binding.gyp 这个重要的配置文件，以及 GYP 文件格式的基础，然后以几个最简单的例子向读者展示了 Node.js 的 C++ 扩展最简单的一些代码，包括函数的参数、回调函数的用法、对象的返回、函数的返回等，以及如何将一个 C++ 的类封装成 Node.js 中直接能用的类。

第 5 章为大家介绍了 NAN (Native Abstractions for Node.js) 这个非常实用的包，使大家能在不同的 Node.js 版本 (本质上是各不兼容的 Chrome V8 版本) 中使用同一份 C++ 代码。

第 6 章讲解了如何使用 libuv 进行异步 Node.js 的 C++ 扩展代码编程，首先介绍了 libuv 的一些基础概念，如句柄与请求等，然后讲述了如何使用 libuv 进行跨线程编程。

第 7 章就开始进入了实战环节。本章通过从零开始写一个基于 C++ 的文件监视器扩展，讲述了要完成一个 Node.js 原生扩展的一些流程。本章所述的文件监视器源码地址在 <https://github.com/XadillaX/node-efsw>。

第 8 章与第 7 章的实战不同，对两个现有的简单 C++ 扩展包进行分析，从另一个角度剖析了一个 Node.js 的 C++ 扩展包的源码。

第 9 章展望了如何使用 Node.js 的最新特性 N-API 进行原生扩展的开发。不过我估计等到本书正式上市的时候，第 9 章已经变成一个仅供参考的章节了。

阅读本书的注意事项

声明：我在编写本书之际，还在大搜车工作，所以书中的很多内容都是基于大搜车的角度来写的。比如 8.2 节中有一处内容是这样的：

在笔者所在公司的内部，用了一套基于 Dubbo 深度定制的 RPC 服务框架。Node.js 要访问这些 Java 服务的 RPC 函数是通过定制的 HTTP 协议来完成的，所有的 RPC 服务节点都到 Zookeeper 进行注册。

这里指的公司就是大搜车。再比如 2.1.2 节中的一段话：

在官方的 Node.js 版本 ONS SDK 出来之前，笔者自己造了一个基于其官方 C++ 版本的 ONS SDK 封装的轮子，用的当然是本书所讲的姿势——Node.js 的 C++ 扩展了。

由于编写本书时我还并未从大搜车离职，因此这仍然是站在就职于大搜车的角度写的。

我在编写本书之际，Node.js 的 8.x 版本并未进入 LTS¹ 阶段。于是我采用了 Node.js 6.x 作为样例进行了讲解，而 Node.js 6.x 距离 LTS 结束也还有一段时间。而且使用本书的方法进行 Node.js 的原生扩展开发，在 Node.js 6.x、Node.js 8.x 甚至是 Node.js 9.x 下都是通用的。本书中的样例都是基于 Node.js v6.9.4 进行讲解的，读者在参考的时候上调或者下调几个中、小版本号问题都不大。

至于 N-API 一章（第 9 章），我在该章中也曾谈道：

本章内容在书中将会一带而过，因为在笔者写书的时候，N-API 还没有完全稳定下来，随时会改变。而且笔者个人认为，距离 N-API 能正式投入生产用途的时间还很长。所以本章内容在本书中仅以扩展阅读的形式存在，其实关于 N-API 的内容在 5.1.2 节中曾略微提及。

因此，该章内容仅供参考，具体内容应以官方文档为准。

另外，本书所有的随书代码均在 macOS 命令行下测试通过。理论上，它们也可以在 Windows 和 UNIX 上运行良好，但我并没有验证过。

最后，给出本书中经常用到的一些地址。

- 本书随书代码的 Git 仓库：<https://github.com/XadillaX/nyaa-nodejs-demo>
- Node.js v6.9.4 代码仓库：<https://github.com/nodejs/node/tree/v6.9.4>
- Node.js v6.x 所对应的 Chrome V8 文档：<https://v8docs.nodesource.com/node-6.12/>。若读者打开该地址，却发现页面不存在，可直接前往 <https://v8docs.nodesource.com/>，并点击“6.x”字样的超链接进入（注意该地址经常换）。
- 作者的个人技术博客：<https://xcoder.in>
- 作者的 GitHub 地址：<https://github.com/XadillaX>
- Me：<https://github.com/XadillaX/me>

¹ <https://github.com/nodejs/Release#release-schedule>。

致谢

感谢我的妻子，她也是一位优秀的 Node.js 研发工程师。她的支持是对我的最大鼓励，如果不是她，这本书的问世也许会更晚。

感谢我的父母，在我的背后默默地支持我的事业。在我很小的时候，他们就一直支持我追寻自己的梦想，这才使我能够在编程领域一路走下来。

感谢我的老东家大搜车，它营造了良好的技术与实践氛围，同事（包括领导）给予了我不少帮助，如书中图示的优化、阅读体验的建议等。这些同事有段鹏飞¹、纪清华、刘佳楠、许波、王琦、袁小山……

感谢现实以及社区中的朋友们在本书创作的时候进行试读和探讨，并提供了一些其他帮助，他们包括但不限于 Akagi201、ADoyle、David Cai、Hax(贺老)、贺星星、精子(jysperm)、孟德森、天然、五花肉、引证、张秋怡。

感谢为本书写序和推荐语的作者们：安娜·亨宁森（Anna Henningsen）、曹力（ShiningRay）、顾天骋（Timothy Gu）、桑世龙（狼叔）、雷宗民（老雷）、刘亚中（Yorkie）、迷渡（justjavac）、潘旻琦（pmq20）、田永强（朴灵）、袁锋（苏千）、孙信宇（芋头）、王文睿博士、响马，你们一直是我们的楷模与学习对象。

感谢我的高中计算机老师兼 NOIP 集训教练王震老师，王震老师是我在编程路上的启蒙老师，没有他就没有今天会写代码的我；也感谢当时陪我坚持走这一条路到毕业的好队友 jiechen 和 MatRush；感谢我的大学 ACM 教练宣江华老师和一直为集训队默默付出的陈萌老师；还要感谢我的研究生导师李启雷博士传道受业。

感谢博文视点的刘皎女士以及她的团队，是他们的努力使本书最终能与广大读者见面，他们提出的专业意见给了我很多帮助。

最后，还要特别感谢董伟明（《Python Web 开发实战》作者）。在阅读了他的一篇文章《写一本技术书籍》²后，我才有了写作本书的想法，并最终付诸实施。

死月（朱凯迪）

2018 年 3 月于杭州

1 以下名字按字典序排序。

2 <https://zhuanlan.zhihu.com/p/22207407>

目录

- 1 Node.js 的 C++ 扩展前驱知识储备 1
 - 1.1 Node.js 的模块机制 2
 - 1.1.1 CommonJS 的模块规范 2
 - 1.1.2 Node.js 的模块 4
 - 1.1.3 小结 9
 - 1.1.4 参考资料 9
 - 1.2 Node.js 的包机制 9
 - 1.2.1 CommonJS 的包规范 9
 - 1.2.2 Node.js / NPM 下的包 13
 - 1.2.3 NPM 与 CNPM 16
 - 1.2.4 小结 19
 - 1.2.5 参考资料 19
 - 1.3 Node.js 依赖简介 20
 - 1.3.1 Chrome V8 20
 - 1.3.2 libuv 25
 - 1.3.3 其他依赖 28
 - 1.3.4 小结 30
 - 1.3.5 参考资料 30
 - 1.4 C++ 扩展开发的准备工作 31
 - 1.4.1 编辑器 / IDE 31
 - 1.4.2 node-gyp 36
 - 1.4.3 其他构建工具 54
 - 1.4.4 小结 56
 - 1.4.5 参考资料 56

2 C++ 模块原理简析	57
2.1 为什么要写 C++ 模块	57
2.1.1 C++ 比 JavaScript 解释器高效	57
2.1.2 已有的 C++ 轮子	72
2.1.3 小结	77
2.1.4 参考资料	77
2.2 什么是 C++ 扩展	78
2.2.1 C++ 模块本质	78
2.2.2 Node.js 模块加载原理	80
2.2.3 小结	102
2.2.4 参考资料	103
3 Chrome V8 基础	104
3.1 Node.js 与 Chrome V8	104
3.2 基本概念	105
3.2.1 内存机制	105
3.2.2 隔离实例 (Isolate)	108
3.2.3 上下文 (Context)	109
3.2.4 脚本 (Script)	110
3.2.5 小结	110
3.2.6 参考资料	111
3.3 句柄 (Handle)	111
3.3.1 本地句柄 (Local)	112
3.3.2 持久句柄 (Persistent)	115
3.3.3 永生句柄 (Eternal)	119
3.3.4 待实本地句柄 (Maybe Local)	119
3.3.5 小结	121
3.3.6 参考资料	121
3.4 句柄作用域	121
3.4.1 一般句柄作用域 (Handle Scope)	122
3.4.2 可逃句柄作用域 (Escapable Handle Scope)	125
3.4.3 小结	129
3.4.4 参考资料	129

3.5	上下文 (Context)	129
3.6	模板 (Template)	133
3.6.1	函数模板 (Function Template)	133
3.6.2	对象模板 (Object Template)	138
3.6.3	对象模板的访问器 (Accessor) 与拦截器 (Interceptor)	144
3.6.4	对象模板的内置字段 (Internal Field)	175
3.6.5	函数模板的继承 (Inherit)	183
3.6.6	小结	188
3.6.7	参考资料	189
3.7	常用数据类型	189
3.7.1	基值 (Value)	189
3.7.2	字符串 (String)	194
3.7.3	数值类型	196
3.7.4	布尔类型 (Boolean)	196
3.7.5	对象 (Object)	196
3.7.6	函数 (Function)	200
3.7.7	数组 (Array)	202
3.7.8	JSON 解析器	203
3.7.9	函数回调信息 (Function Callback Info)	203
3.7.10	函数返回值 (Return Value)	204
3.7.11	隔离实例 (Isolate)	204
3.7.12	小结	205
3.7.13	参考资料	206
3.8	异常机制	206
3.8.1	try-catch	206
3.8.2	抛出异常	209
3.8.3	异常生成类 (Exception)	211
3.8.4	小结	211
3.8.5	参考资料	211
4	C++ 扩展实战初探	212
4.1	binding.gyp	212
4.1.1	惊鸿一瞥	213
4.1.2	binding.gyp 基础结构	213

4.1.3 GYP 文件	214
4.1.4 常用字段	221
4.1.5 小结	228
4.1.6 参考资料	228
4.2 牛刀小试	229
4.2.1 又是 Hello World.	229
4.2.2 函数参数	232
4.2.3 回调函数	234
4.2.4 函数返回	238
4.2.5 小结	239
4.2.6 参考资料	240
4.3 循序渐进	240
4.3.1 C++ 与 JavaScript 类封装	240
4.3.2 实例化 C++ 类封装对象的函数	250
4.3.3 将 C++ 类封装对象传来传去	253
4.3.4 进程退出钩子	255
4.3.5 小结	259
4.3.6 参考资料	259
5 Node.js 原生抽象——NAN	260
5.1 Node.js 原生模块开发方式的变迁	260
5.1.1 以不变应万变	260
5.1.2 时代在召唤	261
5.1.3 小结	267
5.1.4 参考资料	267
5.2 基础开发	267
5.2.1 什么是 NAN	267
5.2.2 安装和配置	269
5.2.3 先睹为快——搭上 NAN 的快车	270
5.2.4 基础帮助函数和宏	276
5.2.5 忽略 node_modules	279
5.2.6 小结	279
5.2.7 参考资料	280

5.3	JavaScript 函数	280
5.3.1	函数参数类型	280
5.3.2	函数声明	282
5.3.3	函数设置	288
5.3.4	小结	296
5.3.5	参考资料	296
5.4	常用帮助类与函数	296
5.4.1	句柄相关	296
5.4.2	创建数据对象	298
5.4.3	与数据对象“玩耍”	300
5.4.4	封装一个类	314
5.4.5	异常处理	315
5.4.6	小结	315
5.4.7	参考资料	316
5.5	NAN 中的异步机制	316
5.5.1	Nan::AsyncQueueWorker	316
5.5.2	Nan::Callback	317
5.5.3	Nan::AsyncWorker	317
5.5.4	Nan::AsyncProgressWorker	323
5.5.5	小结	327
5.5.6	参考资料	327
6	异步之旅——libuv	328
6.1	基础概念	329
6.1.1	事件循环	330
6.1.2	句柄 (Handle) 与请求 (Request)	333
6.1.3	尝尝甜头	335
6.1.4	小结	340
6.1.5	参考资料	340
6.2	libuv 的跨线程编程基础	341
6.2.1	libuv 的线程	342
6.2.2	同步原语 (Synchronization Primitive)	347
6.2.3	工作队列	355
6.2.4	小结	356

6.2.5 参考资料.....	357
6.3 跨线程通信.....	357
6.3.1 uv_async_t 句柄.....	357
6.3.2 Watchdog 半成品实战解析	358
6.3.3 Watchdog 试运行	367
6.3.4 小结.....	368
6.3.5 参考资料.....	369
7 实战——文件监视器.....	370
7.1 准备工作.....	370
7.1.1 功能规划.....	370
7.1.2 文件系统监听库——efsw	373
7.1.3 小结.....	376
7.1.4 参考资料.....	376
7.2 核心设计.....	376
7.2.1 API 设计	377
7.2.2 EFSWCore 的血肉之躯.....	377
7.2.3 EFSWCore 的灵魂.....	381
7.2.4 小结.....	385
7.3 编写 JavaScript 类.....	386
7.3.1 类的设计.....	386
7.3.2 核心逻辑.....	388
7.3.3 简单容错.....	391
7.3.4 小结.....	393
7.4 进一步完善.....	393
7.4.1 C++ 代码的完善.....	393
7.4.2 JavaScript 代码的完善	398
7.4.3 小结.....	400
8 实战——现有包剖析.....	401
8.1 字符串哈希模块——Bling Hashes	401
8.1.1 文件设定.....	402
8.1.2 C++ 源码剖析.....	403
8.1.3 JavaScript 源码剖析	408

8.1.4 小结	409
8.1.5 参考资料	410
8.2 类 Proxy 包——Auto Object	410
8.2.1 Proxy	410
8.2.2 Auto Object 使用范例	412
8.2.3 代码剖析	415
8.2.4 小结	424
8.2.5 参考资料	424
9 N-API——下一代 Node.js C++ 扩展开发方式	425
9.1 浅尝辄止	426
9.1.1 实现一个 Echo 函数	426
9.1.2 尝试运行 N-API 扩展	430
9.1.3 向下兼容	431
9.1.4 N-API Package——C++ 封装	433
9.1.5 小结	433
9.1.6 参考资料	433
9.2 基本数据类型与错误处理	433
9.2.1 基本数据类型	433
9.2.2 与作用域及生命周期相关的数据类型	435
9.2.3 回调数据类型	438
9.2.4 错误处理	439
9.2.5 模块注册	441
9.2.6 小结	442
9.2.7 参考资料	442
9.3 对象与函数	442
9.3.1 对象	442
9.3.2 函数	448
9.3.3 类的封装	453
9.3.4 小结	455
9.3.5 参考资料	455

1

Node.js 的 C++ 扩展前驱知识储备

注：本书所讲解的主要技术就是用 C++ 进行 Node.js 的原生模块（module）开发，其间并不会涉及其他编程语言，所以将在后面以 C++ 模块作为 Node.js 中原生 C++ 模块的简称。

自从 2009 年 5 月，Ryan Dahl 在 GitHub 发布了 Node.js 最初的版本起，经历了几个年的发展，Node.js 目前已经成为计算机技术圈中最火热的技术之一了。

这里暂且不说其效果能与什么抗衡，但借助庞大的生态圈、快速的开发效率，Node.js 已经能涉猎包括但不限于后端、工具脚本¹、GUI 客户端等各种领域。

虽然市面上关于 Node.js 开发的书籍已经非常多了，但是这些图书基本上并未详细涉及用 C++ 进行 Node.js 原生模块开发这个被很多 Node.js 开发者所遗忘的技术。

从本章开始，笔者将逐步扩充读者对于 Node.js 的 C++ 模块开发的前驱知识储备，包括 JavaScript 和 Node.js 的模块机制、包机制，Node.js 的源码依赖和对于开发环境所需要做的准备工作。

¹ 虽然 Node.js 自身与前端并没有直接关系，但是通过 Node.js 实现的各种前端工具链在前端领域中却颇有一席之地。

1.1 Node.js 的模块机制

在介绍 C++ 模块的时候，在此有必要老生常谈，回过头来再讲一遍 Node.js 下的模块机制，这样读者在今后的阅读过程中就能有一个更清晰的思路。

1.1.1 CommonJS 的模块规范

笔者假设读者对 CommonJS 规范、AMD 规范和 CMD 规范等都有了一定的认知。本节我们对 Node.js 所使用的 CommonJS 规范再梳理一遍。首先我们知道 Node.js 的根基就是 JavaScript 或者说是 ECMAScript¹，而 JavaScript 自身是不带模块机制的，所以 CommonJS 规范应运而生，而且 CommonJS 不仅仅是 Node.js 中的模块定义规范。总的来说，它可以给下面的一些场景且不限于这些场景提供相同的规范：

- 服务端 JavaScript 应用；
- 命令行工具；
- 图形化用户界面（Graphical User Interface, GUI）桌面应用；
- 混合应用（如 Titanium²、Adobe AIR 等）。

CommonJS 对于一个模块的定义遵循下面的三个约定。

1. require

`require` 是一个函数，这个函数有一个参数代表模块标识，它的返回值就是其所引用的外部模块所暴露的 API。

讲得直白一点，就是能通过代码 `const biu = require("boom_shakalaka")` 的形式引入 `boom_shakalaka` 这个模块并赋给 `biu`。

2. 模块上下文

在一个 CommonJS 模块的上下文中，需要有满足如下条件的一些事物存在。

- ① `require` 函数，在前面已经提到了。
- ② 一个名为 `exports` 并且长得挺不错的对象，这个 `exports` 对象里面挂载的内容会被暴露到模块外部去。也就是说 `exports` 就是一口通往神魔大陆的“神魔之井”，并且这口连接异界的“神魔之井”也只能叫作 `exports`。

1 有关 ECMAScript 的信息可以参阅 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources。

2 有关 Titanium 的信息可以参阅 https://en.wikipedia.org/wiki/Appcelerator_Titanium。

- ③ 一个名为 `module` 的对象，里面包含了这个模块的一些自带属性，如一个只读的 `"id"` 属性。实际上在 Node.js 中，`module` 对象里面还有一个 `exports` 对象，其初始指针指向与上面说的 `exports` 相同，而且模块真正暴露出去的是 `module` 中的 `exports`。也就是说，如果笔者直接替换了 `module.exports` 对象（如执行代码 `module.exports = {}`），那么导出的就是被替换的对象，而不是上面说的 `exports` 了。至于 `module`、`exports` 与外部模块在 Node.js 模块中的关系，这里用图 1-1 表示。

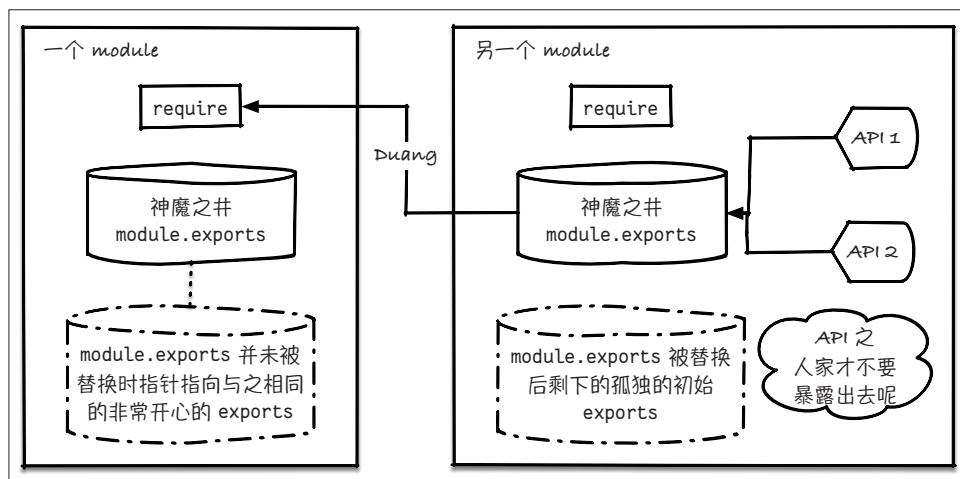


图 1-1 `module`、`exports` 与外部模块的关系

3. 模块标识

模块标识其实就是一个字符串，用于传给 `require` 函数。

它需要小驼峰格式的标识名，或者以 `"."` 以及 `".."` 带头的相对路径。从理论上来说不应该带上后缀名，如 `".js"`。

4. 未指定的约定

这里有两个未指定的约定，也就是说在符合 CommonJS 规范的情况下，无论以下的情况如何都是可用的。

- ① 模块的存储方案未指定，一个模块的内容可以存在于数据库、文件系统、工厂函数，甚至于一个链接库中。
- ② 实现 CommonJS 规范的模块加载器可以支持 `PATH` 环境变量用以加载时的寻径，但是也可以不支持。

4 Node.js: 来一打 C++ 扩展

这里给出一个遵循 CommonJS 规范的简单样例代码¹。

```
// math.js

exports.add = function() {
  var sum = 0, i = 0, args = arguments, l = args.length;
  while(i < l) {
    sum += args[i++];
  }
  return sum;
};
```

```
// increment.js

var add = require('math').add;
exports.increment = function(val) {
  return add(val, 1);
};
```

```
// program.js

var inc = require('increment').increment;
var a = 1;
inc(a); // 2

module.id == "program";
```

1.1.2 Node.js 的模块

Node.js 的应用通过入口文件之后，是由一个个模块组成的。通常一个模块是一个遵循 CommonJS 规范书写的 JavaScript 源文件，也有可能是一个后缀为 *.node 的 C++ 模块二进制文件，这些文件通过 Node.js 中的 `require()` 函数被引入并使用。

使用 CommonJS 规范来作为 Node.js 的模块导出、引入机制，相当于把每个 JavaScript 文件或者模块当作一个不会污染全局的闭包，再配合 NPM²（Node.js 社区目前最流行的包管理系统）2.x 版本的嵌套式依赖方案，能非常优雅地实现某个依赖在一个项目中多版本共存（哪怕是不兼容的多个不同大版本）的情况。与比较适合前端开发的扁平化依赖方案 NPM 3.x 相比，2.x 更适合于 Node.js 应用的开发。

¹ 该样例代码来自 [commonjs.org](http://commonjs.org/wiki/Modules/1.1.1) 的 Wiki，其代表了规范 1.1.1 的样例。CommonJS 规范的更多细节也可以参阅这个网页：<http://wiki.commonjs.org/wiki/Modules/1.1.1>。

² NPM 的官网：<https://www.npmjs.com/>。

1. Node.js 模块寻径

对于 Node.js 来说,除了引入 CommonJS 规范,还对其做了一些附加的工作,比如模块寻径。

在 1.1.1 节中,笔者已经阐述过模块的标识,它是一个遵循小驼峰命名法的字符串,或者是一个以 "."、".." 开头的文件相对路径。但是在 Node.js 中,即使你不遵循小驼峰命名法的规范也是可以的,如 "chinese-random-name" 这个字符串在 Node.js 中也是一个合法的标识。除此之外,它还可以是一个不以 "."、".." 开头的相对路径,甚至也可以是一个绝对路径。

上面几种命名对于 Node.js 来说,会采用几种不同的方法来寻找模块的路径。

(1) Node.js 核心模块

代码存在于 Node.js 源码库,并且将其 API 暴露给开发者的模块称为核心模块。这些核心模块都有自己的预留标识,当执行 `require()` 函数时传入的标识与某个 Node.js 核心模块相吻合时,该函数返回的是该核心模块的 API,如 "fs"、"net" 等。更多核心模块可以参考 Node.js 官方文档¹。

(2) 文件模块

文件模块是指那些需要 Node.js 进行文件寻径获得的模块。下面会阐释寻径方式略微不同的两种模块——**第三方模块**和**项目模块**。

这两种模块虽然寻径方式有略微的不同,但是也有一些共通之处。

如果 Node.js 通过寻径找出的路径代表的是一个目录,那么其会依次寻找该目录下的 "index.js"、"index.json" 以及 "index.node" 文件。若存在上述任何一个字段,则立刻返回。

比如一个 JavaScript 文件路径是 `/Users/biu/index.js`,那么在执行 `require("/Users/biu")` 的时候,该 JavaScript 文件会被加载。

不过寻找到的如果是一个第三方模块目录,其目录下存在一个合法 `package.json` 文件的话,会在上述步骤之前解析 `package.json` 中的 "main" 字段。若该字段存在且合法,那么会直接加载该字段所指向的文件。

如当前目录下有这么一个目录结构,如图 1-2 所示。

如果在 `a_program.js` 中的代码是这样的:

```
// a_program.js
const Biu = require("biu");
```

¹ Node.js v6.9.4 的文档地址是 <https://nodejs.org/dist/v6.9.4/docs/api/>。

6 Node.js: 来一打 C++ 扩展

那么在模块寻径的时候会是 `node_modules/biu` 目录，该目录下同时存在 `package.json` 和 `index.js`，这个时候 Node.js 会先解析 `package.json` 文件。

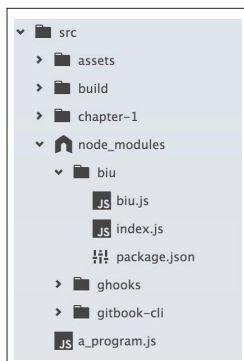


图 1-2 文件模块样例目录结构

这个时候如果 `package.json` 里面是这样的：

```
// package.json 的部分源码

{
  ...
  "main": "biu.js",
  ...
}
```

那么，这个时候执行该 `require()` 所得到的结果就应该是 `node_modules/biu/biu.js` 这个文件了。

(3) 第三方模块

除了上述的核心模块外，其他不是以 `"/"`、`"./"` 或者 `"../"` 开头的字符串作为标识的模块被称为第三方模块，这些模块通常以 Node.js 依赖包的形式存在。当 `require()` 函数传入的是第三方模块的标识时，则 Node.js 不仅仅在当前目录的 `node_modules` 目录下寻找文件名或者文件夹名与之相吻合的模块。这个“不仅仅”的意思如下：

- ① 当前文件目录的 `node_modules` 目录下；
- ② 若 ① 没有符合的模块，则去当前文件目录的父目录的 `node_modules` 下；
- ③ 若没有符合的模块，则再往上一层目录的 `node_modules`；
- ④ 若没有符合的模块，重复 ③，直到寻找到符合的模块或者根目录为止。

在找到符合的模块之后，`require()` 函数就会返回找到的模块所暴露的 API 了。

(4) 项目模块

在一个项目中执行 `require()` 来载入一个 `"/"`、`"/."` 或者 `"/../"` 带头的模块就是项目模块了。它会加载文件路径相对于传入的标识的相对路径的模块，或者是绝对路径所指向的模块。由于通常加载 Node.js 模块的时候我们并不需要给其加上后缀名，这也是 CommonJS 中所规定的，因此 Node.js 在加载项目模块或者第三方模块的时候会枚举尝试后缀名。尝试的后缀名依次为 `".js"`、`".json"` 和 `".node"`，其中 `".node"` 就是 C++ 模块。

图 1-3 显示了一个项目模块的样例目录结构。

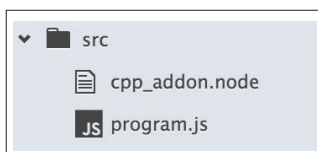


图 1-3 项目模块样例目录结构

假设当前目录下的文件结构如图 1-3 所示，如果我们在 `program.js` 的代码是这样的：

```
const CPP = require("../cpp_addon");
```

那么在模块加载寻径的时候，显而易见 `../cpp_addon` 是不存在的，这个时候 Node.js 就会依次去寻找 `../cpp_addon.js`、`../cpp_addon.json` 和 `../cpp_addon.node`，最后会加载并返回 `cpp_addon.node` 所暴露的 API。

2. 模块缓存

实际上在 Node.js 运行中，通常情况下一个包一旦被加载了，那么在第二次执行 `require()` 的时候就会在缓存中获取暴露的 API，而不会重新加载一遍该模块里面的代码再次返回。

如果有一个文件是 `dog.js`，代码如下¹：

```
// dog.js

"use strict";

let boom = "嘘，蛋花汤";
boom += " 在睡觉。🐶";

module.exports = {
```

¹ 蛋花汤是笔者养的一条宠物狗的名字。

8 Node.js: 来一打 C++ 扩展

```
"🐶": boom
};
```

而且在相同目录下有一个 `entry.js`，代码如下¹：

```
// entry.js

"use strict";

let 🐶_在睡觉_🐶 = require("./dog");

console.log(🐶_在睡觉_🐶);

let 蛋花汤 = require("./dog");

console.log(蛋花汤);
```

在 `entry.js` 中执行 `require()` 时会通过寻径找到 `dog.js` 并执行，暴露出来的 `🐶` 变量值为 "嘘，蛋花汤在睡觉。🐶"，也就是说输出会如下所示：

```
$ node entry.js
{ '🐶': '嘘，蛋花汤在睡觉。🐶' }
{ '🐶': '嘘，蛋花汤在睡觉。🐶' }
```

第一句输出 `{ '🐶': '嘘，蛋花汤在睡觉。🐶' }`，理所当然；到了第二句的 `require()` 时，由于 `dog.js` 已经被加载过了，Node.js 中会将其暴露的内容缓存到它的运行时中（其原理会在后续的章节中加以阐释），这个时候再执行 `require()` 就会直接返回内存中已加载好的 `module.exports`，如下所示：

```
{
  "🐶": boom
}
```

而不会出现重新执行一遍 `dog.js`，重新声明 `boom` 这个变量，更不会在原来的 `boom` 基础上再拼接一次 "在睡觉。🐶" 以及出现 "嘘，蛋花汤在睡觉。🐶 在睡觉。🐶" 的荒谬结果。

1 在 Node.js 以及大多数的 JavaScript 解释器中都支持 Unicode 字符串作为变量名，可以参照 <https://mathiasbynens.be/notes/javascript-identifiers>。当然，在实际线上项目中笔者还是推荐按照团队规范来，而个人的玩具项目则可以多尝试这些好玩的特性。

1.1.3 小结

本节主要讲述了 CommonJS 的模块规范，以及基于它进行改装的 Node.js 模块规范。

本节列举了 CommonJS 模块规范的三个要素：`require`、模块上下文和模块标识，讲述了 Node.js 的模块寻径算法和模块缓存机制。

这些老生常谈的基础内容是在 Node.js C++ 模块开发中必不可少的，且不仅仅局限于 C++ 模块开发，任何 Node.js 模块开发都离不开这些基础知识。

1.1.4 参考资料

[1] Modules/1.1.1: <http://wiki.commonjs.org/wiki/Modules/1.1.1>.

[2] 朴灵. 深入浅出 Node.js[M]. 北京: 人民邮电出版社, 2013.

[3] Modules - Node.js v6.9.4 Documentation: https://nodejs.org/docs/v6.9.4/api/modules.html#modules_addenda_package_manager_tips.

1.2 Node.js 的包机制

在 1.1 节中笔者讲述了 Node.js 中以文件作为粒度的模块机制，接下来要讲的是一个或者多个模块结合起来的 Node.js 包机制。

1.2.1 CommonJS 的包规范

CommonJS 规范中的包 (package) 指的是一系列模块、代码以及其他资源文件的一个封装。它为 CommonJS 输出内容的交付、安装、管理等提供了一个方便的途径。

一个 CommonJS 包必然含有一个包描述文件，以及可选地包含了一些代码、资源等内容。本书中讲解的 CommonJS 规范为 1.0 规范。

1. 包描述文件

一个遵循 CommonJS 规范的包必然包含一个包描述文件，并且它处于包根目录下，名为 `package.json`。

在这个 JSON 文件中指定了这个包的一些必要信息，如表 1-1 所示。

表 1-1 CommonJS 包描述文件的必填字段

字段名	类型	含义
name	字符串	包的名字，并且是一个仅由小写字母、数字、_、- 以及 . 组成的唯一字符串
description	字符串	包的简要描述。理论上包描述的第一句话（到第一个 . 之前的内容）用作包列表显示时的标题
version	字符串	版本号，须遵循语义化版本规范 ^①
keywords	字符串数组	包的关键词。数组里面的每一个字符串代表一个包的关键词，理论上用于包的搜索
maintainers	对象数组	包的维护者。数组里面的每一个对象代表一个维护者，并且每一个对象的格式为 { "name": "死月", "email": "i@2333.moe", "web": "http://xcoder.in/" }。其中 name 为必要字段，其余两个字段为可选字段
contributors	对象数组	包的贡献者。数组里面的每一个对象代表一个贡献者，格式与 maintainers 一致
bugs	字符串	代表提交 Bug 用的链接，可以为 http(s) 协议或者 mailto 协议
licenses	对象数组	该包的许可协议（通常用于开源）。一个对象代表一个许可，并且每一个对象的格式为 { "type": "GPLv2", "url": "http://www.example.com/licenses/gpl.html" }。其中 type 表示许可名（如 GPLv2、MIT ^② ）等，url 表示许可正文的地址
repositories	对象数组	源码仓库。数组里面的每一个对象代表一个源码仓库，并且每一个对象的格式为 { "type": "git", "url": "http://github.com/example.git", "path": "packages/mypackage" }。其中，type 代表仓库类型（如 "git"），而 url 代表仓库地址。如果该包不在源码仓库的根目录中，还可以指定一个 path 字段来指明目录
dependencies	对象	包依赖。一个 CommonJS 包可以依赖一个或多个其他 CommonJS 包。在该对象的键值对中，键名代表依赖包名，键值表示版本；当然，也有更复杂一些的指定方法 ^③

① 有关语义化版本规范的信息可以参阅 <http://semver.org/>。

② 有关官方开源许可的更多信息可以参阅 <https://opensource.org/licenses/alphabetical>。

③ 有关更复杂的依赖指定格式可参阅 CommonJS 的包规范 Wiki: <http://wiki.commonjs.org/wiki/Packages/1.0>。

除了上述的必填字段之外，官方还指定了一些有用的选填字段，如表 1-2 所示。

表 1-2 CommonJS 包描述文件的选填字段

字段名	类型	含义
homepage	字符串	包的主页地址
os	字符串数组	包可用的操作系统。一个字符串代表一个支持的操作系统，若为空则代表所有操作系统都支持，其中操作系统需要为这些值：aix、freebsd、linux、macos、solaris、vxworks、windows
cpu	字符串数组	包可用的 CPU 架构。一个字符串代表一个支持的 CPU 架构，若为空则代表所有 CPU 都支持，其中 CPU 需要为这些值：arm、mips、ppc、sparc、x86、x86_64
engine	字符串数组	包可用的 JavaScript 解释引擎。一个字符串代表一个支持的引擎，若为空则代表所有引擎都支持，其中引擎需要为这些值：ejs、flussspferd、gpsee、jsc、spidermonkey、narwhal、node、rhino、v8
builtin	布尔	是否为底层平台内置的包
directories	对象	包目录映射。包管理器执行时通过键名来寻找目录，如 { "lib": "src/lib" }
implements	字符串数组	每个字符串代表该包支持的相关 CommonJS 规范，如 CommonJS/Modules/1.0
scripts	对象	包脚本映射，用于包管理，如 { "install": "install.js" }

注意：遵循 CommonJS 规范的包管理器和运行程序会忽略包描述文件中的未知字段名（即未在上述规范中声明的字段），这些字段可以让开发者定义一些个性化的内容。

但是需要注意的是还有一些字段虽然未在上面列出，但是作为保留字段有可能会在未来被使用，开发者在定义个性化字段的时候应当避免使用这些保留字段。

保留字段如下：build、default、email、external、files、imports、maintainer、paths、platform、require、summary、test、using、downloads、uid、type。

下面给出一个简单的遵循 CommonJS 包规范的描述文件。

```
{
  "name": "mypackage",
  "version": "0.7.0",
  "description": "Sample package for CommonJS. This package demonstrates the
```

12 Node.js: 来一打 C++ 扩展

```
required elements of a CommonJS package.",
  "keywords": [
    "package",
    "example"
  ],
  "maintainers": [
    {
      "name": "Bill Smith",
      "email": "bills@example.com",
      "web": "http://www.example.com"
    }
  ],
  "contributors": [
    {
      "name": "Mary Brown",
      "email": "maryb@embedthis.com",
      "web": "http://www.embedthis.com"
    }
  ],
  "bugs": {
    "mail": "dev@example.com",
    "web": "http://www.example.com/bugs"
  },
  "licenses": [
    {
      "type": "GPLv2",
      "url": "http://www.example.org/licenses/gpl.html"
    }
  ],
  "repositories": [
    {
      "type": "git",
      "url": "http://hg.example.com/mypackage.git"
    }
  ],
  "dependencies": {
    "webkit": "1.2",
    "ssl": {
      "gnutls": ["1.0", "2.0"],
      "openssl": "0.9.8"
    }
  },
  "implements": ["cjs-module-0.3", "cjs-jsgi-0.1"],
  "os": ["linux", "macos", "win"],
  "cpu": ["x86", "ppc", "x86_64"],
  "engines": ["v8", "ejs", "node", "rhino"],
```

```

"scripts": {
  "install": "install.js",
  "uninstall": "uninstall.js",
  "build": "build.js",
  "test": "test.js"
},
"directories": {
  "lib": "src/lib",
  "bin": "local/binaries",
  "jars": "java"
}
}

```

2. 包格式

CommonJS 包的格式是一个包含整个包目录(尤其是 `package.json` 文件)的 ZIP 格式压缩包,不过其有可能在未来版本的规范中有所不同。

3. 包目录结构

一个遵循 CommonJS 规范的包目录特点如下:

- `package.json` 在根目录下;
- 二进制文件应当在 `bin` 目录下;
- JavaScript 源码应当在 `lib` 目录下;
- 文档应当在 `doc` 目录下;
- 单元测试文件应当在 `test` 目录下。

1.2.2 Node.js / NPM 下的包

NPM 在最开始的时候是 Node.js Package Manager 的缩写,即 Node.js 包管理器。后来因为前端也开始使用 NPM 进行包管理,所以它的意义就开始改变了,变成了 JavaScript 包管理器,其结构和原理也有了一些明显的改变。

1. 包的路径和依赖

在 1.1 节中我们提到过,当 Node.js 在项目中执行 `require()` 并传入的是一个非路径类的字符串时,会按照层级在各 `node_modules` 文件夹下寻找相应的文件或者文件夹。

实际上 Node.js 的第三方包都放在该目录,而在项目根目录中会有一个 `package.json` 文件记录该项目会有哪些第三方包的依赖。

虽然大家用法各异，有人甚至在 Git 版本库中也会将该文件夹放进去，并且肆意修改 `node_modules` 目录下的源码，但实际上这样的方式是非常不值得推荐的。

通常的做法是在项目 `package.json` 中写上依赖信息，并在线上机器（或者部署机器）上再执行 `npm install` 进行依赖安装。如果有真的需要修改第三方库代码的，推荐在项目运行初始化或者合适的时机对它进行 `hack`。

如果一个项目按照规范的写法去做，那么在项目目录下的 `node_modules` 目录被称为第三方包目录，也就是第三方开发者（也有可能是你自己）写的包会被安装到里面去。

2. NPM 下的包描述文件

NPM 中的项目目录和包的根目录下都需要有一个 `package.json` 文件来做一些定义。这里的 `package.json` 大体遵循 CommonJS 规范，但是与 CommonJS 的包规范相比，仍有比较多的不同之处。

注意，笔者在后面并不会特别强调 Node.js 项目的 `package.json` 文件与 CommonJS 规范相比有比较多的不同之处并加以解释。因为在实际应用中，Node.js 在对包进行引入的时候对 `package.json` 中的相关信息依赖较少，反而是对 `node_modules` 这个目录更为依赖。开发者甚至只要是在 `node_modules` 目录下新建一个文件或者文件夹，并于其中新建一个 `*.js` 的源码文件，如 `node_modules/foo/index.js`，那么在项目中就可以直接通过 `require("foo")` 来引入这个模块了。

实际上，笔者在后面会强调的是 `package.json` 文件对于 NPM 使用的影响。由于 Node.js 开发中最常用的包管理系统就是 NPM，因此经常在说明的时候会将两者等价起来。

换言之，如果哪天有人出了一个新的 Node.js 可以使用的包管理器，其需要其他样式的包描述文件，那么使用它管理包的时候就不一定再是本书中解说的 `package.json` 了。事实上 Node.js 中流行的包管理器的确不止 NPM 一种，就现在来说还有阿里巴巴 Node.js 生态中的 CNPM¹、Facebook 出品的 Yarn²，只不过就目前而言其包描述文件基本上与 NPM 一致。

这里讲讲 NPM 下的 `package.json` 与 CommonJS 规范下的 `package.json` 主要的几个不同点（对其部分字段进行比较），如表 1-3 所示。

1 CNPM 的更多信息可以参阅 <https://cnpmjs.org/package/cnpm>。

2 Yarn 的更多信息可以参阅 <https://yarnpkg.com/lang/en/>。

表 1-3 NPM 中 package.json 与 CommonJS 包规范中 package.json 的部分字段比较

字段名	类型	含义
name	字符串	该字段与 CommonJS 规范相比主要做了字数、规则上的限制。如最大长度为 214，可以有 @SCOPE/ ^① 前缀等
bugs	对象	该字段类型与 CommonJS 规范略有不同，是一个包含 "url" 字段和 "email" 字段的对象
license	字符串	该类型与 CommonJS 不同。这里的 license 是一个字符串，可以是 SPDX 许可中的一个名字，也可以是类似于 "SEE LICENSE IN <filename>" 这样的字符串
author	字符串 / 对象	该字段对应于 CommonJS 中的 maintainers，并且只有一个人。它可以是一个字符串（形式如 "名字 <邮箱> (主页地址)"），或者与 maintainers 中的元素差不多的对象，其区别在于表示网站的字段从 web 变成了 url
contributors	字符串数组 / 对象数组	该字段与 CommonJS 略不同，其数组中的每一个元素格式都与 author 中每一个元素的格式相同
files	字符串数组	包中包含的文件。不声明该字段则会包含所有未被忽略的文件，否则会包含该字段所声明的文件，其余文件会被忽略
main	字符串	包的入口文件，主要用于定位执行 require("包") 时寻址用。若未指定，那么引入该包时默认会引入包根目录下的 index.js 文件，否则会在包根目录下根据该字段声明的文件名来引入
bin	字符串 / 对象	主要用于定义包安装后的可执行文件名
repository	字符串 / 对象	其为对象时与 CommonJS 相比少了一个 path 字段。其为字符串时只要用于定义 GitHub、GitHub gist、BitBucket、GitLab 等仓库，如 "XadillaX/thmclrx"，就代表了仓库 https://github.com/XadillaX/thmclrx
dependencies	对象	与 CommonJS 相比，该字段只能是一级的对象，不能有嵌套行为，并且键值除了可以是语义化的版本号以外，还可以用 ^、~ 等符号来做版本号的上下浮动，也可以定义版本号的范围或一个 tag，甚至可以是一个文件名、包的压缩包名以及一个 Git 仓库地址

续表

字段名	类型	含义
devDependencies	对象	其格式与 dependencies 一致，代表开发时的依赖，会在包根目录执行 <code>\$ npm install</code> 时安装这些依赖，而通过 <code>\$ npm install 包名</code> 安装某个包时不会在该包目录下安装这个包的开发依赖
engines	对象	与 CommonJS 一样，表示支持该包的引擎。不过其格式与 dependencies 类似，表示指定某个引擎的版本号范围，如 <code>{ "engines" : { "node" : ">=0.10.3 <0.12" } }</code>
os	字符串数组	与 CommonJS 一样，表示支持该包的操作系统，不过它还可以用逻辑运算符非进行表示，如 <code>"os" : ["!win32"]</code>
cpu	字符串数组	与 os 相同，可以用非进行表示，如 <code>"cpu" : ["!arm", "!mips"]</code>

① `@SCOPE/` 前缀主要用于包的命名空间，实现隔离效果。如大搜车公司内部有个私有包 `confbiu`，那么我们就可以将这个包命名为 `@souche/confbiu`。

其他更多的信息读者可以自行去 NPM 官网中查阅¹。

1.2.3 NPM 与 CNPM

本节对 NPM 与 CNPM 的分析，主要是针对它们的包安装目录结构的不同进行的。

由于本书主要写的内容是 Node.js 的原生依赖开发，因此下面给出一些针对 Node.js 开发者的建议，并简述了一些与前端开发依赖的区别。

1. NPM

NPM 最初是由 Isaac Z. Schlueter 开发，用于进行 Node.js 的依赖包发布、管理和安装的。后来由于前端包管理系统 Bower 可能无法满足前端开发人员的一些需求，因此很多前端开发人员也开始用 NPM 管理前端的包依赖。不过，原来 NPM 的路径形式不是特别适合前端的开发，因此自从 NPM 3 开始，其安装形式就发生了一些变化。该变化除了为了适应突如其来的前端开发需求之外，还有一部分原因是为了解决 Windows 下开发时文件路径过长的问题²。

1 NPM 中的 package.json 字段说明可以参阅 <https://docs.npmjs.com/files/package.json>。

2 关于 NPM 在 Windows 开发环境下路径过长的情况，可以参阅 GitHub 上相关的 Issue: <https://github.com/npm/npm/issues/3697>。

(1) NPM 2

NPM 的一个非常重要的分水岭在于版本 2 和版本 3。

NPM 2 的依赖安装路径是嵌套式 (Nested) 的, 非常适合 Node.js 进行工具、后端的开发。

下面举个例子, 我们的项目有一个依赖包 `bar` 的 1.0.0 版本依赖另一个包 `foo` 的 ^1.0.0 版本, 而我们项目的另一个依赖包 `baz` 的 1.0.0 版本依赖了 `foo` 的 ^1.1.0 版本, 那么通过 NPM 2 进行安装的包依赖目录就会如下所示:

```

├── node_modules
│   ├── bar
│   │   └── node_modules
│   │       └── foo
│   └── baz
│       └── node_modules
│           └── foo

```

其中 `bar` 和 `baz` 的代码里面执行 `require("foo")` 的时候分别引入的是不同的 `foo` 依赖副本, 这样在它们中间就不会产生一些破坏性的关系。

举个简单的例子, 如果 `bar` 的开发者觉得 `foo` 中有一个函数无法满足自己的需求, 那么也许它会有这么一段代码:

```

const foo = require("foo");

const old = foo.func;
foo.func = function() {
  // do some hack
  // 做一些注入式的代码, 使其满足自己的要求
  // 但这段代码可能会引起其他使用该包的依赖
  // 造成破坏
};

```

这个时候如果使用 NPM 2 进行开发的话没有什么太大的问题, 除非这个项目本身也引用了 `foo` 这个依赖, 导致 `bar` 和 `baz` 指向的都是项目本身的 `foo` 路径。

但是如果使用 NPM 3 的话, 代码会陷入一个不可预知的危险环境中。

所以通常来说, 如果环境允许, 并且你是一位 Node.js 开发者的话, 还是推荐使用 NPM 2 进行包依赖, 并且摒弃那些只支持 NPM 3 的包。当然, 这种主观意愿还是因人而异的, 这里只是笔者的一个建议而已。

(2) NPM 3+

这里的 3+ 指的是 NPM 3 及其以上版本, 并且有可能一直包括其未来的所有版本。

它的依赖安装模式是扁平化（Flatten）的，非常适合前端项目：需要优化到很小的代码空间占用量、打包的分析等；同时也解决了 Windows 中文件路径不允许过长的的问题。

在 NPM 3 发布时的更新日志（Changelog）中我们能发现，它一再强调了扁平化这个特性。

你的依赖会被尽可能地扁平化。所有你的依赖、你依赖的依赖以及每一级的依赖都会被安装到你项目的根 `node_modules` 目录下，不再嵌套。只有在两个依赖存在冲突的情况下才会出现嵌套的情况。¹

也就是说它在解决省了一点点空间的问题、Windows 安装路径过长的问题时，又带入了另一个冲突不可预知的问题。不过这种情况在前端开发中可以忽略不计，我们可以放心使用。并且在巨大的包量冲击下，一大拨新版本的依赖都开始使用了这个危险的特性，导致其开始不兼容 NPM 2，使得很多 Node.js 开发者不得不转而使用 NPM 3。

比如你在使用 NPM 2 安装一个依赖的时候，由于其依赖的依赖不会被安装到根 `node_modules` 下，导致其有些地方在执行 `require` 的时候找不到相应的包。

还有一个问题就是当你安装了一个项目的依赖的时候，会出现一个比较尴尬的情景：明明你自己项目的依赖可能只有十几个，但是在你的 `node_modules` 目录里很有可能会出现一两千个文件夹。这对于可能需要经常打开 `node_modules` 来调试一些依赖包的代码的开发者来说无疑是一种折磨。

不过这也许只是笔者个人的感受，如果读者对上述的一些问题着实无感的话，可以忽略上面的一些建议。

2. CNPM

CNPM² 是阿里巴巴集团的 Node.js 团队开发的一套安装更快的 NPM 工具，其 Logo 如图 1-4 所示。它由苏千³和死马⁴主导，内核使用了其自主研发的 `npminstall`⁵。



图 1-4 CNPM 的 Logo

1 <https://github.com/npm/npm/releases/tag/v3.0.0>

2 CNPM 的 GitHub 地址是 <https://github.com/cnpm/cnpm/>。

3 苏千，真名袁锋，蚂蚁金服 Node.js 技术专家。

4 死马，真名何翊宇，蚂蚁金服 Node.js 技术专家，Koa 的维护者。

5 `npminstall` 的 GitHub 地址是 <https://github.com/cnpm/npminstall/>。

就算不使用国内的镜像，CNPM 的依赖安装速度仍远远大于 NPM。它的工作原理是，将一些包缓存到 `node_modules/.npminstall` 目录下，再以符号链接（Symbol Link）的形式将依赖目录连接到其对应的路径。这样相同版本的包在安装过程中实际上只有一份实体。

对于 CNPM 来说，除了上面所说的不同于 NPM 的点之外，它的分水岭在于 CNPM 4.2 以及 4.3+。

在 CNPM 4.2 的版本里面，除去符号链接的特性外，它的目录结构与 NPM 2 一致，是嵌套式的。

后来由于 NPM 3 的影响实在太太大，导致很多前端依赖以及部分 Node.js 依赖在 NPM 2 下无法正常工作，因此 CNPM 在 4.3+ 的版本中为前端开发者加入了扁平化的支持。

不过为了使 CNPM 不仅仅服务于前端开发者，CNPM 还结合了嵌套式依赖的特性。也就是说它除了完完全全按照 CNPM 4.2 版本的嵌套式目录结构安装了依赖之外，还会顺便将一些通过计算得到的所有依赖以及所有依赖的依赖的特定版本文件放了一份到 `node_modules` 目录下。这既解决了前端开发者的需求，又保留了原始 Node.js 开发者的需求。

不过，`node_modules` 下目录成群的尴尬情景就成了无法解决的死结。

1.2.4 小结

本节主要讲了 CommonJS 规范下包的定义、入口文件和目录结构，以及其与 NPM 包管理程序的异同。Node.js 主要使用 NPM 的一套规范来使用第三方依赖。

在 1.2.3 节中讲述了 NPM 2 与 NPM 3 的异同，并阐述了两个不同版本针对 Node.js 开发和前端开发的友好度的异同。这里推荐了既满足 Node.js 开发者又满足前端开发者需求的 CNPM。

这在之后的 Node.js 原生依赖开发中会向开发者提供一些目录结构上的引导，并且希望读者自己写的依赖同时依赖 NPM 2 和 NPM 3。不要过于依赖其嵌套式或者扁平化的特性，以致其在某些情况下出现不能使用的情景。

1.2.5 参考资料

- [1] Packages/1.0: <http://wiki.commonjs.org/wiki/Packages/1.0>.
- [2] package.json, Specifics of npm's package.json handling: <https://docs.npmjs.com/files/package.json>.
- [3] Force npm 3 to install a node module always nested: <https://github.com/npm/npm/issues/9809>.
- [4] npminstall: <https://github.com/cnpm/npminstall>.

1.3 Node.js 依赖简介

从某种意义上来说，Node.js 并不是一个从零开始编写的 JavaScript 运行时，它其实也是站在“巨人的肩膀”上进行了一系列的拼凑和封装得到的结果。它的高效离不开一些很牛的第三方程序和类库。

1.3.1 Chrome V8

Chrome V8 简称 V8，是由谷歌开源的一个高性能 JavaScript 引擎。该引擎采用 C++ 编写，Google Chrome 浏览器用的就是这个引擎。V8 可以单独运行，也可以嵌入 C++ 应用当中。其 Logo 如图 1-5 所示。



图 1-5 V8 的 Logo

和其他的 JavaScript 引擎一样，V8 会编译、执行 JavaScript 代码，并一样会管理内存、垃圾回收等。

就是因为 V8 的高性能以及跨平台等特性，所以它也是 Node.js 的 JavaScript 引擎。

1. 高效

V8 开发小组由一群程序语言专家组成。其中核心工程师 Lars Bak 之前在 Sun 公司工作，专注于 Java 虚拟机加速技术的研究，产出了 HotSpot，除此之外，他还曾开发了 Strongtalk¹。

所以，V8 的代码里面蕴含了从 HotSpot 和 Strongtalk 中汲取的精髓。

该研发小组从 2006 年开始研发 V8，原因是当年市面上的各种 JavaScript 引擎效率都比较低下。在 Lars Bak 等人的贡献下，JavaScript 引擎添加了新的一员——Chrome V8，并且效率非常高。

V8 的高效主要体现在以下 4 个特性上面。

¹ Strongtalk 项目主页：<http://www.strongtalk.org/>。

(1) JIT 编译

JIT 编译, 全称 Just-In-Time 编译, 也就是即时编译。它编译出的结果直接是机器语言, 而不是字节码。这样大大提高了 V8 在执行 JavaScript 时的效率。不过后来其他的几家 JavaScript 引擎也渐渐推出了对 JIT 的支持。

(2) 垃圾回收

这个特性在 Java 领域中使用得比较多。虽然其他语言或者其他的 JavaScript 引擎实现都有垃圾回收, 但是 V8 的垃圾回收借鉴了 Java VM 的精确垃圾回收管理, 而其他很多语言的垃圾回收用的是保守垃圾管理。

相较而言, V8 的这套垃圾回收机制的效率要远远高于其他一些垃圾回收机制实现——实际上代价就是这种机制的实现难度更大。

(3) 内联缓存 (Inline Cache)

V8 使用了内联缓存的特性来提高属性的访问效率。如有一个访问是 `this.蛋花汤`, 没有内联缓存的时候, 每次要取蛋花汤的话都会对哈希表进行一次寻址, 而加入了内联缓存的特性之后, V8 能马上知道这个属性的一个偏移量, 而不用再次计算寻址的偏移量了。

(4) 隐藏类

由于 JavaScript 是一门动态的编程语言, 因此哪怕是在 ES6 及以上版本的规范中有了 `class` 的一个定义, 开发者也能非常方便地对一个对象添加或者移除一个属性。

隐藏类就是对这样一套对象体系中的一个黑科技的包装——所有如属性一样的对象会被归为同一个隐藏类。

下面举个简单的例子:

```
function Pet(type, name) {
  this.type = type;
  this.name = name;
}

var 蛋花汤 = new Pet("🐶", "蛋花汤");
var 南瓜饼 = new Pet("🥮", "南瓜饼");
蛋花汤.age = 4;
```

一开始根据 `Pet` 创建了 `蛋花汤` 这个对象。在最开始初始化的时候 V8 就会创建一个隐藏类 (假设是 `P0`), 这是一个空类, 因为它还没有任何的属性; 后来 `this.type = type` 执行了, 隐藏类就有了 `type` 属性, 这个时候就又多了一个 `P1` 的隐藏类——`P1` 是基于 `P0` 创建的, 并且多了 `type` 属性; 接着, `name` 被赋值上去, 于是隐藏类又多了一个 `P2`。

然后在创建 南瓜饼 对象的时候，又走了上面的老路，只不过这次不是创建隐藏类 P0、P1 和 P2 了，而是直接沿用它们。在初始化 南瓜饼 的时候，它依次会属于上面创建的 3 个隐藏类，直到最后它跟 蛋花汤 一样都属于 P2。

最后一行代码在给 蛋花汤 赋值 age 的时候，又一个新的隐藏类 P3 会被创建。这个时候 蛋花汤 和 南瓜饼 分别属于 P3 和 P2。这些描述分别如图 1-6、图 1-7、图 1-8 和图 1-9 所示。

隐藏类和内联缓存这两把“匕首”联合起来，是 V8 高效的一个非常重要的原因，因为同一个隐藏类的对象们能用同一套内联缓存来寻址。

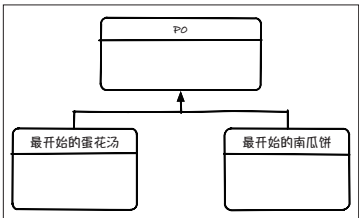


图 1-6 最开始的蛋花汤和南瓜饼隐藏类归属

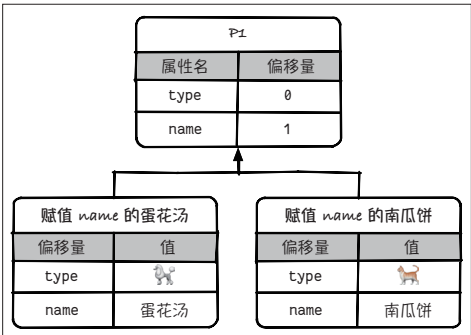


图 1-7 赋值 type 后的蛋花汤和南瓜饼隐藏类归属

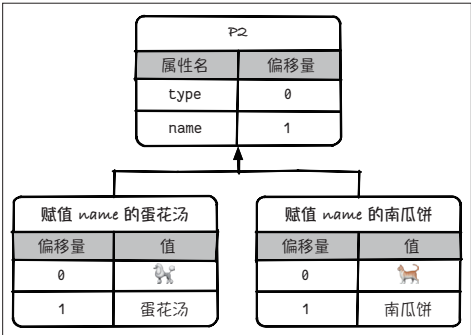


图 1-8 赋值 name 后的蛋花汤和南瓜饼隐藏类归属

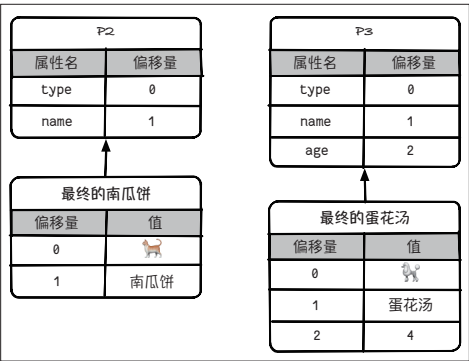


图 1-9 最终的蛋花汤和南瓜饼隐藏类归属

2. 遵循 ECMAScript

在当前 V8 的项目主页¹中，有一句话表明了它是遵循 ECMA-262 标准的：

“V8 implements ECMAScript as specified in ECMA-262.”

就目前来说 ECMA-262 标准（曾）发布了 7 个大版本，如表 1-4 所示。

表 1-4 ECMA-262 标准的 7 个大版本

版本	发布日期	与之前版本对比的修改	编者
1	1997 年 6 月	第一个版本	Guy L. Steele Jr.
2	1998 年 6 月	格式修正，使其完全符合 ISO/IEC 16262 国际标准	Mike Cowlishaw
3	1999 年 12 月	添加正则表达式；更好的词法作用域链处理；新的控制指令；try-catch 支持；错误定义更加明确；数据输出的格式化以及其他变化	Mike Cowlishaw
4	-	第 4 版的“小可爱”被委屈地遗弃掉了，因为产生了语言复杂性方面上的一些分歧。不过还好，这个“小可爱”为第 5 版 Harmony 提供了部分基础	
5	2009 年 12 月	添加了“strict 模式”；锐化了第 3 版中的一些模糊规范；增加了一些新特性，如 getter 和 setter，以及 JSON 库的支持等	Pratap Lakshman, Allen Wirfs-Brock
5.1	2011 年 6 月	ECMAScript 5.1 标准与第 3 版 ISO/IEC 16262:2011 国际标准完全吻合	Pratap Lakshman, Allen Wirfs-Brock

¹ Chrome V8（简称 V8）项目主页：<https://developers.google.com/v8/>。

续表

版本	发布日期	与之前版本对比的修改	编者
6	2015 年 6 月	原名 ECMAScript 6（或 ES6），后改名为 ECMAScript 2015（或 ES2015）；添加了很多新的语法，如类（class）、模块、for/of 循环、菊花函数 ^① 、箭头函数等	Allen Wirfs-Brock
7	2016 年 6 月	第 7 版也被称为 ECMAScript 2016，继续改造语法和语法糖等	Brian Terlson

① Generator 函数的爱称，因其有一个显著的标识——形如菊花的星号（*）而得名。

V8 在开发的过程中也一直追着 ECMAScript 发布的脚步，如基本上完成了对 ES6 的支持，而且最新版也对 async/await 函数进行了支持。¹

也正是因为 V8 对 ECMAScript 标准紧追不舍，才有了 Node.js 能及时跟上 ECMAScript 最新语法的情况。如 Node.js 7.6 正式默认支持 async/await 功能就是沾了 V8 的光。

3. Node.js 与 Chrome V8

表 1-5 是 V8 与 Node.js 的部分版本对照表。

表 1-5 V8 与 Node.js 的部分版本对照表

V8 版本	Node.js 版本区间	V8 版本	Node.js 版本区间
4.5.103.30	v4.0.0	4.5.103.36	v4.4.6, v4.4.7
4.5.103.33	v4.1.0, v4.1.1	4.5.103.37	v4.5.0 - v4.6.1
4.5.103.35	v4.1.2 - v4.4.5	4.5.103.42	v4.6.2
4.5.103.43	v4.7.0 - v4.7.3	5.1.281.84	v6.8.0 - v6.9.1
4.5.103.45	v4.8.0	5.1.281.88	v6.9.2
4.5.103.46	v4.8.1	5.1.281.89	v6.9.3 - v6.9.5
4.6.85.28	v5.0.0	5.1.281.93	v6.10.0
4.6.85.31	v5.1.0 - v5.11.1	5.1.281.95	v6.10.1
4.6.85.32	v5.12.0	5.4.500.36	v7.0.0, v7.1.0
5.0.71.35	v6.0.0, v6.1.0	5.4.500.43	v7.2.0

¹ 有关 Chrome V8 的最新资讯（如版本发布等）可以参考其博客：<https://v8project.blogspot.com>。

续表

V8 版本	Node.js 版本区间	V8 版本	Node.js 版本区间
5.0.71.47	v6.2.0	5.4.500.44	v7.2.1
5.0.71.52	v6.2.1 - v6.3.0	5.4.500.45	v7.3.0, v7.4.0
5.0.71.57	v6.3.1	5.4.500.48	v7.5.0
5.0.71.60	v6.4.0	5.5.372.40	v7.6.0
5.1.281.81	v6.5.0	5.5.372.41	v7.7.0 - v7.7.3
5.1.281.83	v6.6.0, v6.7.0	5.5.372.42	v7.7.4

由表 1-5 可见，Node.js 一直紧跟 V8 的版本脚步在迭代。

Node.js 与 V8 实际上看起来更像是一对情侣，而不仅仅是 Node.js 一厢情愿地使用 V8 作为自己的底层支持。

在 Chrome V8 的博客中曾经有一篇文章名为《V8 ♥ Node.js》¹。Node.js 在几年发展中的流行度稳步增长，于是有了 V8 的“姑娘，你成功引起了我的注意”。现在 V8 也有一些工作是为 Node.js 而做的：

- 在 Chrome 开发者工具中可以调试 Node.js；
- 加速 ES6；
- 针对 Node.js vm 模块和 REPL 的一些修复；
- Async / await。

1.3.2 libuv

libuv 是一个专注于异步 I/O 的跨平台类库。实际上它主要为 Node.js 而开发，不过也被其他的一些程序使用，如 Luvit²、Julia³、pyuv⁴ 等⁵。其 Logo 如图 1-10 所示。

¹ 《V8 ♥ Node.js》文章地址：<https://v8project.blogspot.com/2016/12/v8-nodejs.html>。

² Lua + libuv + JIT 的缩写，其是 Node.js 的 Lua 实现，可参考 <https://luvit.io>。

³ Julia 语言，其是一门高层次的、高性能的动态编程语言，可参考 <http://julialang.org/>。

⁴ libuv 的 Python 接口实现，可参考 <https://pyuv.readthedocs.io/>。

⁵ 更多使用了 libuv 的程序可以参考这张列表：<https://github.com/libuv/libuv/wiki/Projects-that-use-libuv>。



图 1-10 libuv Logo

Node.js 的一个重要概念就是事件循环。而 Node.js 的事件循环就是由这个 libuv 进行驱动的。有了 libuv 对 Node.js 的事件循环支持，读者才能在 Node.js 的异步世界中自由翱翔。

1. 亮点特性

libuv 的异步中有非常多的亮点，让开发者甚至不需要自己去管理线程等内容就能轻松地实现一套异步代码：

- 基于 `epoll` / `kqueue` / `IOCP` / `event ports` 实现的全能事件循环；
- 异步 TCP 和 UDP 套接字；
- 异步 DNS 解析；
- 异步文件、文件系统操作；
- 文件系统事件；
- ANSI 转义码控制的 TTY；
- 使用 UNIX domain 套接字或者命名管道实现的套接字共享 IPC；
- 子进程；
- 线程池；
- 信号（Signal）处理；
- 高精度时钟；
- 线程和同步元。

正是不同的平台有不同的异步机制（如 `epoll`、`IOCP` 等），libuv 才能基于它们实现跨平台的事件循环，Node.js 才能尝到甜头。

2. 接口简单

老实说，对于一个别人写的库，我爱不爱用主要是考察其 API 设计如何。也就是该怎么用，设计得好不好，有没有冗余设计。文档之类的对于我来说不太重要，反正有代码可以看嘛。

对于 libuv 我大体上还算满意，用 C 实现可以加很多分。

因为我觉得一个库，若想被人当成黑盒子去使用，以后也被当成黑盒子来维护，甚至可以用别的盒子去替代它，关键的一点就是接口简单。这个简单包括了使用最少的概念、需要最少的知识去理解它。

——节选自云风¹的博客中的《libuv 初窥》，这也是笔者非常赞同的一点。

libuv 的接口非常简单，并且明了。辅以少量的文档就能快速地上手。这是作为一个给别人使用的类库所拥有的很棒的品格。

这对本书的读者来说也是一种莫大的幸运——因为你在学习 C++ 模块开发的时候，在 libuv 这一部分只需要花非常少量的时间就能上手了。

3. libuv 与 Node.js

libuv 是 Node.js 最初的作者 Ryan Dahl 为 Node.js 写的底层异步库。所以可以说它天生就是为 Node.js 而生的，在流行起来之后也为其他很多项目提供了基础的“血液”。

Node.js 的事件循环直接用的就是 libuv 的事件循环。在 Node.js v6.9.4 下，src/node.cc 文件里的 4690 到 4698 行代码中²我们能发现它把 libuv 的默认事件循环传入 NodeInstanceData 供其执行了。

```
NodeInstanceData instance_data(NodeInstanceType::MAIN,
                                uv_default_loop(),
                                // 这个函数就是获取 libuv 的默认事件循环
                                argc,
                                const_cast<const char**>(argv),
                                exec_argc,
                                exec_argv,
                                use_debug_agent);
StartNodeInstance(&instance_data);
exit_code = instance_data.exit_code();
```

¹ 云风，真名吴云洋，曾任网易杭州研究中心总监，是《大话西游》《梦幻西游》等的主要开发者，后自己创业，任新游戏公司 CTO。他在编程和游戏开发领域是非常有名的工程师。

² 此处提到的代码出处如下：<https://github.com/nodejs/node/blob/v6.9.4/src/node.cc#L4690-L4698>。

其实不仅仅是 6.9.4 版本，其他版本的 `src/node.cc` 文件中也能找到对应的相关代码（即使它看起来跟上面的代码不太一样）。

1.3.3 其他依赖

本节会介绍 Node.js 除了 Chrome V8 引擎和 libuv 以外的一些依赖。如果读者想了解它的全部依赖，可以去翻看 Node.js 源码的 `deps` 目录。

1. http-parser

这是一个 C 实现的 HTTP 消息解析器，其能解析 HTTP 协议的请求数据和返回数据。它的项目地址在 <https://github.com/nodejs/http-parser>，由此可见 `http-parser` 是 Node.js 项目抽象出来的一个第三方库，主要也是为 Node.js 提供相应的功能。

它的特性如下：

- 无依赖；
- 持久化连接的流式处理（keep-alive）；
- 分段信息（chunk）的解码；
- 缓冲区溢出攻击的防御。

在上述特性的基础上，该依赖主要解析 HTTP 消息中下面的一些内容：

- 消息头键值对（Header）；
- 内容长度（Content-Length）；
- 请求方法（GET、POST、PUT、DELETE 等）；
- 返回状态码（Status Code）；
- 传输编码（Transfer-Encoding）；
- HTTP 版本；
- 请求地址（URL）；
- 消息体。

其实不仅仅是在 Node.js 中，如果开发者自己在开发 C / C++ 项目时有相关需求，也可以直接引用这个库。

2. OpenSSL

OpenSSL 是一套大名鼎鼎的安全套接字层协议库。

SSL 就是 Secure Socket Layer 的缩写，其可以在网络上提供秘密性传输。Netscape 公司¹在推出第一个 Web 浏览器 Netscape Navigator²的同时提出了 SSL 的标准，用于对 HTTPS 协议进行加密。SSL 包含记录层（Record Layer）和传输层，记录层协议确定传输层数据的封装格式。传输层安全协议使用 X.509³ 认证，之后利用非对称加密演算来对通信方做身份认证，之后交换对称密钥作为会谈密钥（Session key）。这个会谈密钥用于将通信两方交换的数据做加密，保证两个应用间通信的保密性和可靠性，使客户与服务器应用之间的通信不被攻击者窃听。

OpenSSL 主要由 C 语言编写，实现了基本的加密功能：SSL 与 TLS 协议。同样，OpenSSL 也是跨平台的。

Node.js 在安全性相关层面的代码就基于 OpenSSL 进行了一个封装，如 HTTPS 协议支持。

扩展阅读：即便是 OpenSSL 这样一个强大的开源库，也不是完全无懈可击的。在 2014 年 4 月 7 日，一个关于 OpenSSL 的“心脏出血漏洞”（Heartbleed bug）被公开披露。该漏洞的成因是由于在实现 TLS 心跳扩展时没有对输入进行适当验证，导致实际可以读取的数据比允许读取的数据要多。不过在该漏洞被公开披露的当天，OpenSSL 发布了修复后的版本。

3. zlib

zlib 是一个老牌的年代久远的提供数据压缩功能的库，由 Jean-loup Gailly 和 Mark Adler 开发。它最初是为 libpng⁴ 库所写的，不过后来普遍被许多软件所使用。有趣的是，Jean-loup 负责 zlib 的压缩逻辑而 Mark Adler 负责解压。其 Logo 如图 1-11 所示。



图 1-11 zlib Logo

在 Node.js 中免不了对一些数据进行压缩、解压处理，很多时候就是用 zlib 的一个封装实现的。

1 其中文名为网景，这是一个自 1994 年开始的品牌，曾是一家美国的计算机服务公司，以其生产的同名 Web 浏览器而闻名。

2 其又被称为网景浏览器，或者网景导航者，是网景公司开发的 Web 浏览器。

3 在密码学中，X.509 是由 ITU-T 为了公开密钥基础设施建设（PKI）与授权管理基础建设（PMI）而提出的产业标准。

4 用于 PNG 图形格式的一个实现，对 bitmap 数据规定了 DEFLATE 作为流压缩方法。

扩展阅读：关于 zlib 的各种资料读者可以自行搜索，这里稍微提一下一个由 Facebook 开发的另一套更高效的压缩算法——Zstandard（简称 zstd）。其官方主页是 <http://facebook.github.io/zstd/>。

在其官方主页中有一张性能对比表格，其中能看出 Zstandard 的成绩遥遥领先于其他一些类库。Zstandard 与其他压缩库的性能对比如表 1-6 所示。

表 1-6 Zstandard 与其他压缩库的性能对比

名字 / 版本	压缩比	压缩速度	解压速度
zstd 1.1.3 - 1	2.877	430 MB/s	1110 MB/s
zlib 1.2.8 - 1	2.743	110 MB/s	400 MB/s
brothli 0.5.2 - 0	2.708	400 MB/s	430 MB/s
quicklz 1.5.0 - 1	2.238	550 MB/s	710 MB/s
lzo1x 2.09 - 1	2.108	650 MB/s	830 MB/s
lz4 1.7.5	2.101	720 MB/s	3600 MB/s
snappy 1.1.3	2.091	500 MB/s	1650 MB/s
lzf 3.6 - 1	2.077	400 MB/s	860 MB/s

1.3.4 小结

本节主要介绍了 Node.js 与其他一些第三方库的不解之缘。Node.js 是社区贡献者们站在巨人肩膀上所完成的一个结果。

Chrome V8 为其提供了 JavaScript 解释和运行时的引擎，libuv 为它的事件循环提供了非常好的载体。http-parser、OpenSSL、zlib 等第三方库都成为 Node.js 的枝丫。

正是有了这些巨人，才有了现在无论是开发还是执行都效率满满的 Node.js。

1.3.5 参考资料

[1] 为什么 V8 引擎这么快？： <http://blog.csdn.net/horkychen/article/details/7761199>.
[2] Just-in-time compilation： https://en.wikipedia.org/wiki/Just-in-time_compilation.
[3] Javascript Hidden Classes and Inline Caching in V8： <http://richardartoul.github.io/jekyll/update/2015/04/26/hidden-classes.html>.

- [4] ECMAScript: <https://en.wikipedia.org/wiki/ECMAScript>.
- [5] Node 7.6 Brings Default Async/Await Support: <https://www.infoq.com/news/2017/02/node-76-async-await>.
- [6] V8 ♥ Node.js: <https://v8project.blogspot.com/2016/12/v8-nodejs.html>.
- [7] Previous Releases | Node.js: <https://nodejs.org/en/download/releases/>.
- [8] libuv 初窥: <http://blog.codingnow.com/2012/01/libuv.html>.
- [9] Transport Layer Security: https://en.wikipedia.org/wiki/Transport_Layer_Security.
- [10] X.509: <https://en.wikipedia.org/wiki/X.509>.
- [11] zlib Home Site: <http://www.zlib.net/>.

1.4 C++ 扩展开发的准备工作

前面讲了一些概念性的内容，本节将介绍入手一个 Node.js 的 C++ 扩展时要做的一些准备工作。

1.4.1 编辑器 / IDE

“工欲善其事，必先利其器”，有一个称心的编辑器或者 IDE 是非常重要的事情。这里推荐几款比较强大的工具。

1. VIM / NeoVIM

VIM 是从 vi¹ 发展出来的一个文本编辑器，在配置完备、插件附体的情况下其功能可以媲美于一款 IDE，在程序员中被广泛使用。其和 Emacs 并列成为类 UNIX 系统用户最喜欢的编辑器。VIM 的第一个版本由 Bram Moolenaar 在 1991 年发布，最初是 Vi IMitation 的简称。不过随着后来其功能不断增加，正式的名称改成了 Vi IMproved。

NeoVIM 是对 VIM 的一个重构版本，也可以说是 VIM 的一个超集，它兼容 VIM 的配置，但是也有自己的一些配置策略。

该项目自 2014 年发起，就目前来说已经可以非常方便地使用了。

¹ vi 是一种计算机文本编辑器，由美国计算机科学家比尔·乔伊 (Bill Joy) 编写，于 1976 年发表，并以 BSD 授权发布。

不过对于暂时还没有 VIM 基础的读者来说，一开始入门的学习曲线会比较陡峭，但是一旦学成之后就可以“肆无忌惮”了。其玩笑化的学习曲线如图 1-12 所示。

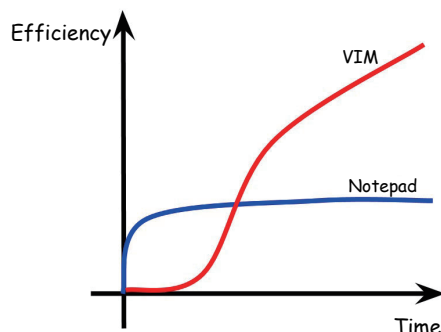


图 1-12 网络上关于 VIM 学习曲线的玩笑

若想学习 VIM，除了搜索网上的教程以外，还可以去阅读 O'Reilly 系列图书《学习 vi 和 Vim 编辑器》的原版或者中文版。

想要尝试 NeoVIM 的读者可以参考 NeoVIM 官网 Wiki 上的安装一节：<https://github.com/neovim/neovim/wiki/Installing-Neovim>。

在安装完 VIM 或者 NeoVIM 之后，你可能需要一款强大的 VIM 下的代码补全插件来助你在 C++ 世界中厮杀，这个插件就是 YouCompleteMe (<https://github.com/Valloric/YouCompleteMe>)。另外，你还需要一款语法检查插件——syntastic (<https://github.com/vim-syntastic/syntastic>)。

2. Emacs

Emacs（源自 Editor MACroS，宏编辑器），是一个文本编辑器家族，具有强大的可扩展性，在程序员和其他以技术工作为主的计算机用户中广受欢迎。其最初由 Richard Stallman 于 1975 年在 MIT 协同 Guy Lewis Steele Jr. 共同完成。

网上广为流传的一句话“Emacs 其实是一个操作系统”足以证明其很强大。人们用 Emacs 不仅可以写代码，还能管理文件系统、收发邮件、上网听音乐等。

想用 Emacs 开发的读者可以上它的官网 (<https://www.gnu.org/software/emacs/>) 了解更多的信息。

3. Sublime Text

Sublime Text 是一款具有代码高亮、语法提升、自动补全等功能的编辑器，其界面美观，同样有着插件扩展机制，这套机制可以基于 Python 进行开发。经过一系列发展之后，Sublime Text 已经由 2 升级至 3。

Sublime 具有下面的一些特性：

- 快速跳转（文件、符号或者行数）；
- 指令框（快速输入指令）；
- 多行选择；
- 基于 Python 的插件机制；
- 项目单独配置；
- JSON 格式的配置文件；
- 跨平台。

在集成了一堆插件之后，其同样是一款功能强大的编辑器。图 1-13 为笔者从网络上摘录的一张 Sublime Text 截图。

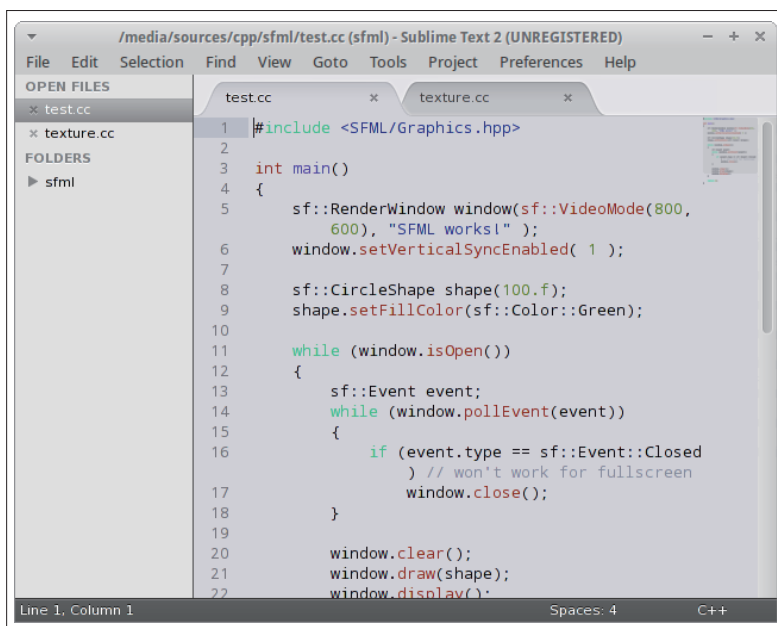


图 1-13 Sublime Text 截图

想用 Sublime Text 开发的读者可以上它的官网（<https://www.sublimetext.com/>）了解更多信息。

4. Visual Studio Code

Visual Studio Code 是一款比较新潮的编辑器，由 Microsoft 公司出品。其跨平台且开源，对 Node.js（调试）、TypeScript、Go 等语言的支持较好，并且性能非常高。

中国 Node.js 领域的布道者之一 i5ting¹ 曾撰写了一份关于 Visual Studio Code 的指南(Visual Studio Code Guide, <http://i5ting.github.io/vsc/>)，里面详细介绍了 Visual Studio Code 的各种用法。图 1-14 为笔者从网络上摘录的一张 Visual Studio Code 截图。

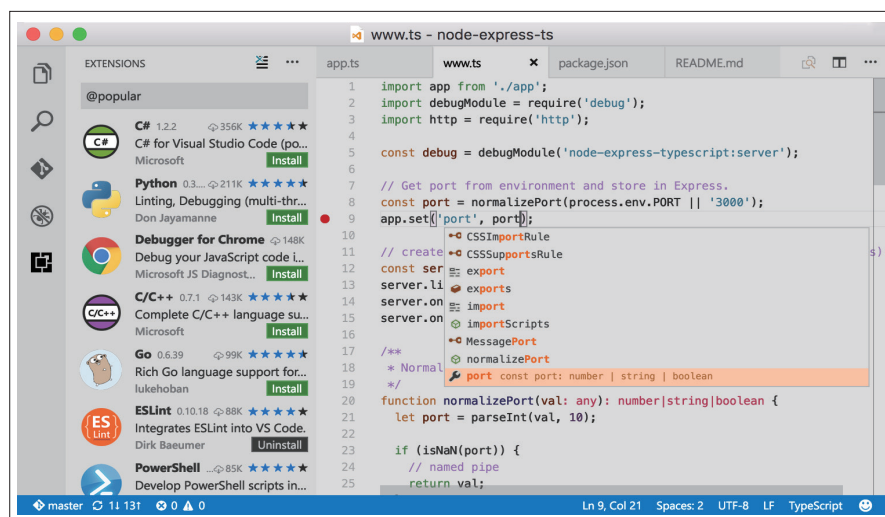


图 1-14 Visual Studio Code 截图

想用 Visual Studio Code 开发的读者可以上它的官网（<https://code.visualstudio.com/>）了解更多信息。

5. Atom

Atom 是 GitHub 在 2014 年发布的一款基于 Web 技术构建的文本编辑器，其底层基于 GitHub 自己开源的 Electron² 开发。

1 其原名为桑世龙，在网络上多以狼叔、i5ting 的名字出现，全栈工程师，是国内 Node.js 的布道者之一。

2 一个能用 JavaScript、HTML 和 CSS 开发跨平台桌面应用的框架。

Atom 有什么亮点呢？我总结了这样几点：

- 像 Sublime Text 一样开箱即用；
- 像 Emacs 一样允许开发者充分地定制；
- 基于 JavaScript 和 Web 技术构建；
- 开源且拥有一个活跃的社区。

——摘自 jysperm 博客的《Atom 背后的故事》
详见 <https://jysperm.me/2016/11/behind-atom/>

上面这段节选总结起来就是 Atom 易用、高可扩展。

想用 Atom 开发的读者可以上它的官网（<https://atom.io/>）了解更多信息。

6. Visual Studio

Visual Studio 是由 Microsoft 公司开发的一款 Windows 系统下强大的 IDE。它包含了整个软件生命周期所需要的大部分工具，如 UML、代码管控等。而且其对于 C++ 的支持异常强大，可以说是世界上最好的 IDE 之一。

虽然 Visual Studio 已经非常老牌了，但是它一直与时俱进，其最新的版本照样有对于 Node.js 的支持。并且其还有一个令人兴奋的点是，Microsoft 已经着手 Visual Studio 对于 macOS 系统的支持，目前已经有针对 macOS 的版本发布。

在 Windows 下用 Visual Studio 还有一个理由，就是如果用 node-gyp¹ 形式管理 Node.js C++ 模块的话，在 Windows 系统下 Visual Studio 是必要的依赖。在编译时的中间步骤中，node-gyp 生成的中间目录中就有 Visual Studio 专用的项目文件（*.sln）。这个中间文件可以直接使用 Visual Studio 打开进行项目编写和调试。

如果读者觉得麻烦，也可以用其他的编辑器进行编码，只是 Windows 用户在编译 C++ 模块的时候 Visual Studio 是必须要安装的，或者至少要安装其编译工具²。图 1-15 为笔者从网络上摘录的一张 Visual Studio 截图。

¹ node-gyp 在较新版本中的 Node.js 都是自带的，用来编译原生的 C++ 模块。在后续的章节中会对 node-gyp 进行详细介绍。

² 即 Visual C++ 构建工具（Build Tools），可以访问 <http://landinghub.visualstudio.com/visual-cpp-build-tools> 获取更详细的信息。

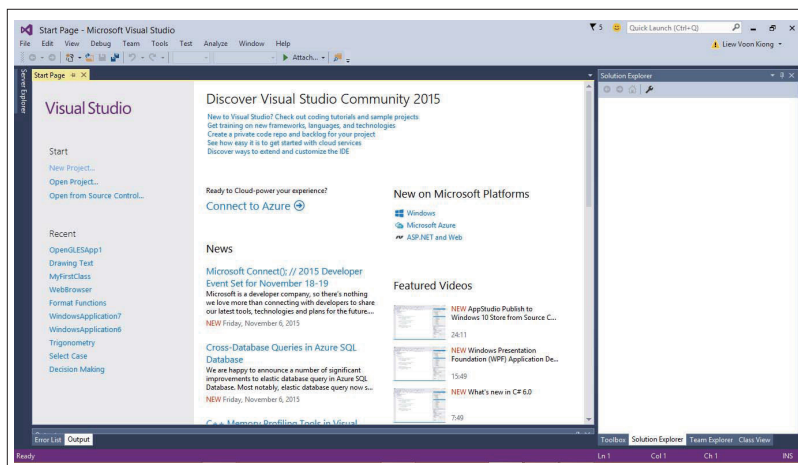


图 1-15 Visual Studio 截图

想用 Visual Studio 开发的读者可以上它的官网（<https://www.visualstudio.com/zh-hans/>）了解更多信息。

1.4.2 node-gyp

node-gyp 是 Node.js 下的 C++ 扩展构建工具。几乎所有的 Node.js C++ 扩展都是由它构建的¹。

它基于 GYP 来进行工作。GYP 的全称为 Generate Your Projects，它是谷歌出品的一套构建工具，通过一个 *.gyp 的描述文件生成不同系统所需要的项目文件（如 Makefile 或者 Visual Studio 项目文件）以供构建和编译。

使用 node-gyp 的 C++ 扩展中主要由一个 binding.gyp 文件进行配置。

较新的 Node.js 版本中会自带 node-gyp。当然，如果没有自带或者需要升级，我们也能自行安装。

1. 依赖

在安装 node-gyp 之前，我们还需要安装一些它所需要的依赖，才能供其正常工作。其中每个平台都需要的依赖就是 Python 2.7。

注意：一定是 Python 2.7，因为 node-gyp 在 Python 3.x 下并不能很好地工作。

¹ “几乎所有”，也就是说其实还是有例外的。node-gyp 是目前 Node.js 中最流行的 C++ 扩展构建工具，但是也有一些其他工具，如 cmake-js（<https://www.npmjs.com/package/cmake-js>）等，不过这些工具不在本书的详细介绍范围之内。

(1) 如果读者用的是 UNIX

在 UNIX 系列平台下工作的读者大概需要这三样东西：

- Python 2.7（通常系统自带）；
- `make`¹；
- C++ 编译工具包（如 GCC²）。

以 Ubuntu 14.04 为例，从完全没有上述依赖的情况开始，大概需要通过下面的几条命令来安装。

• Git

在本书的讲解中，Git³ 是比较重要的一个工具，所以推荐读者事先安装好 Git。

```
$ [sudo] apt-get install git
```

• Python

```
$ [sudo] apt-get install python2.7
```

在接下来的输入提示中看情况去区分 `y` 或者 `n` 即可。通常情况下全部选 `y` 选项。

不过通常对于像 Ubuntu 这类 Linux 发行版来说，Python 2.7 是自带的，不需要再安装。

• `make` 和 `gcc`

在 Ubuntu 的最精简版本安装时，`make` 和 `gcc` 是不随系统安装的。如果你的环境中已经存在着 `make` 和 `gcc`，则可以略过本步骤。

我们可以一步安装这些与构建相关的程序集：

```
$ [sudo] apt-get install -y build-essential
```

如果是其他的发行版，则可以单独安装这些程序。

1 即 GNU `make`，它是一个工具程序，用于自动化构建软件，通常在 UNIX 系列系统下运行。可以访问 <https://www.gnu.org/software/make/> 获取更多信息。

2 即 GNU Compiler Collection，它包含了 C、C++ 等一系列语言的编译工具。

3 Git 是一款免费、开源的分布式版本控制系统，用于敏捷、高效地处理任何或小或大的项目。

(2) 如果读者用的是 macOS

在 macOS 下搭建 node-gyp 的环境依赖大体与 UNIX 下相同，只不过它的构建工具源于 Xcode¹。

• Xcode

Xcode 里面包含了各种构建所需要的工具，如 GCC 等。

首先需要从 App Store 下载安装 Xcode。事先已经安装了 Xcode 的读者可以略过这一步。

图 1-16 为 Xcode 在 App Store 中的安装界面。



图 1-16 Xcode 的安装界面

在安装好 Xcode 后，需要通过它的入口来安装命令行工具（Command Line Tools）。事先已经通过 Xcode 安装命令行工具的读者可以略过这一步。

打开 Xcode，然后依次点击菜单栏的 Xcode → Preferences → Downloads 进行下载。

(3) 如果读者用的是 Windows

在 Windows 系统下，微软的 Windows Build Tools 是必备的前驱依赖。这些依赖可以通过安装 Visual Studio 2015（或其他可用版本）、Visual C++ Build Tools 或者是 NPM 中的 windows-build-tools 来获得。

¹ Xcode 是运行在操作系统 macOS 上的集成开发工具（IDE），由苹果公司开发。

- Visual Studio

有些读者可能在先前的学习或者工作经验中就已经接触过 Visual Studio，所以其计算机中已有这样的 IDE，这样就可以略过本节以及与 Windows Build Tools 相关的章节。

如果你并没有 Windows Builds Tools 中的任意一项，并且想以安装 Visual Studio 的方式获取的话，就可以阅读一下本节。

首先前往 <https://www.visualstudio.com/zh-hans/> 或者通过搜索引擎下载到 Visual Studio 2015（或其他可用版本），然后执行安装步骤。图 1-17 和图 1-18 分别为 Visual Studio 安装时不同步骤的界面。

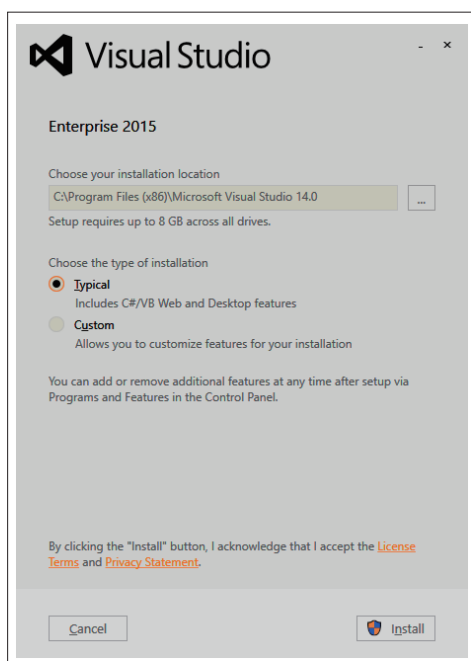


图 1-17 Visual Studio 安装步骤 1

然后在下一步的安装界面中选择与 Visual C++ 相关的选项。

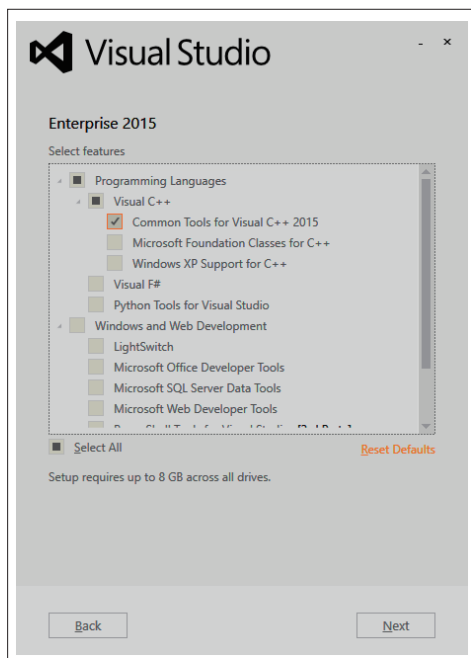


图 1-18 Visual Studio 安装步骤 2

安装完毕重启即可。最后要在 NPM 中设置一下 Visual Studio 的版本号。执行下面的命令：

```
C:\>npm config set msvs_version 2015
```

注意：如果你使用 Visual Studio 2015 进行安装，则上述命令后面的版本号不需要改；如果是其他 node-gyp 支持的版本，则只需要把后面的版本改成相应的版本即可。

• Visual C++ Build Tools

如果开发者在开发过程中并不需要使用 Visual Studio 这么庞大的 IDE，也可以使用稍微轻量一点的只包含 Visual C++ 的构建工具包来完成依赖。

要安装 Visual C++ Build Tools，可以前往其相关的官网：<http://landinghub.visualstudio.com/visual-cpp-build-tools>。

图 1-19 为 Visual C++ Build Tools 官网的截图，图 1-20 为其安装界面截图。

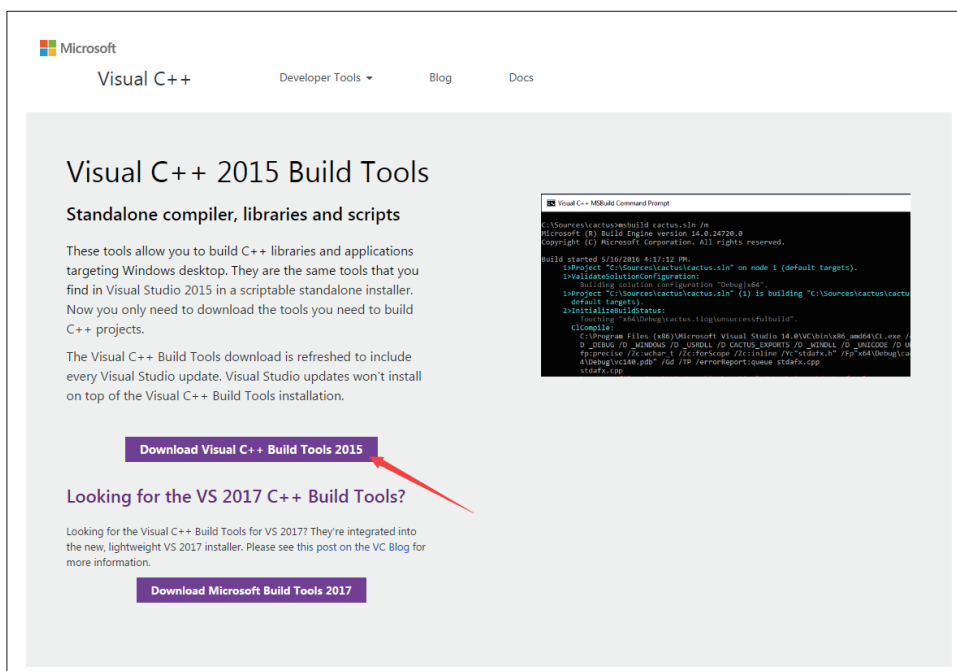


图 1-19 Visual C++ Build Tools 官网

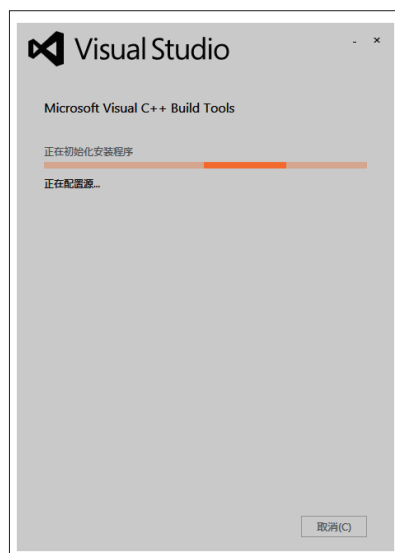


图 1-20 Visual C++ Build Tools 的安装界面

点击了 Download Visual C++ Build Tools 2015 之后就会跳出下载弹窗。等下载好之后就可以进行安装了。

根据安装界面上引导的步骤进行安装，安装完成，之后重启即可。然后在 NPM 中设置一下 Visual Studio 版本号：

```
C:\>npm config set msvs_version 2015
```

• NPM 的 windows-build-tools

如果直接使用 NPM，也可以安装在 NPM 官方仓库中的 windows-build-tools 依赖。其包名就是 windows-build-tools，更多信息可以访问它在 GitHub 上的仓库进行了解：<https://github.com/felixrieseberg/windows-build-tools>。

在安装 windows-build-tools 之前需要保证你的计算机中已经安装了 .NET Framework 4.5.1。若你的计算机尚未安装 .NET Framework 相应版本的话，可以前往 <https://www.microsoft.com/en-us/download/details.aspx?id=40773> 进行下载并且安装。

若要安装 windows-build-tools，请打开你的 Windows 终端软件（如 Command、PowerShell、Cygwin 等），然后执行下面的命令：

```
C:\>npm install --global --production windows-build-tools
```

可能出现的结果会跟下面的输出比较像。

```
> windows-build-tools@1.2.0 postinstall C:\Users\NODEJS\AppData\Roaming\npm\node_modules\windows-build-tools
> node ./lib/index.js

Downloading BuildTools_Full.exe
[=====>] 100.0% of 3.29 MB (2.63 MB/s)
Downloading python-2.7.11.msi
[=====>] 100.0% of 18.64 MB (241.25 kB/s)

Downloaded python-2.7.11.msi. Saved to C:\Users\NODEJS\.windows-build-tools\python-2.7.11.msi.
Starting installation...
Launched installers, now waiting for them to finish.
This will likely take some time - please be patient!
Waiting for installers... |Successfully installed Visual Studio Build Tools.
Waiting for installers... |Successfully installed Python 2.7
windows-build-tools@1.2.0 C:\Users\NODEJS\AppData\Roaming\npm\node_modules\windows-build-tools
```

```

|—— cli-spinner@0.2.6
|—— debug@2.6.3 (ms@0.7.2)
|—— fs-extra@1.0.0 (jsonfile@2.4.0, graceful-fs@4.1.11, klaw@1.3.1)
|—— chalk@1.1.3 (escape-string-regexp@1.0.5, supports-color@2.0.0, ansi-
styles@2.2.1, has-ansi@2.0.0, strip-ansi@3.0.1)
|—— nugget@2.0.1 (throttleit@0.0.2, minimist@1.2.0, single-line-log@1.1.2,
progress-stream@1.2.0, pretty-bytes@1.0.4, request@2.81.0)

```

由于上述命令要前往微软官网、Python 官网等下载相应的程序，因此在国内的网络中下载会有速度缓慢或者丢包等现象，请耐心等待或重试。

执行了上述命令后，如果一切顺利，那么你就完成了 node-gyp 前驱依赖 windows-build-tools 的安装，里面包含了 Visual C++ Build Tools 和 Python 2.7。

• Python

node-gyp 除了需要 Visual C++ 的构建工具之外，还需要 Python 来支持其正常工作。如果你使用了 NPM 的 windows-build-tools 包安装方式来安装 node-gyp 前驱依赖的话，可以省去这一步，因为它自带了 Python。但如果你使用另两种方式安装 Visual C++ 构建工具的话，还是要安装一下 Python 2.7 的。

首先，前往 <https://www.python.org/downloads/> 进行下载并安装。

注意：下载 Python 2.7.* 版本，node-gyp 不支持 Python 3+。

图 1-21 为在 Windows 下安装 Python 2.7 的界面截图。

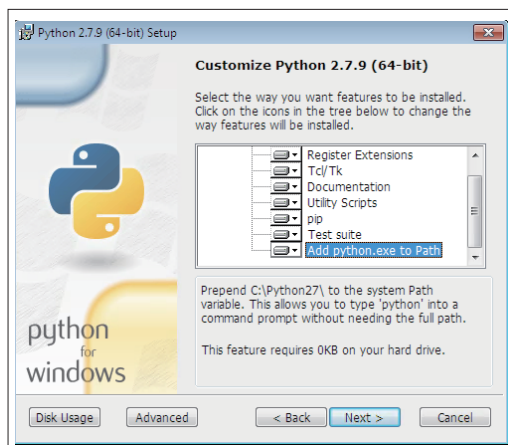


图 1-21 Windows Python 2.7 的安装界面

注意要将 `python.exe` 添加到环境变量中。安装界面中有这一选项，选择就可以了。安装完成之后重启计算机。

然后在 NPM 中设置一下 Python 版本即可：

```
C:\>npm config set python python2.7
```

如果你的计算机中安装了多个版本的 Python 的话，可执行下面的命令指定相应的 Python：

```
C:\>npm config set python <PYTHON 目录>
```

• Git

为了方便地跟随本书的脚步，对于在 Windows 下开发的用户笔者推荐安装 Git¹。如果本机没有安装 Git 环境，请读者自行前往 <https://git-scm.com> 下载 Git 并安装。图 1-22 为 Windows 下的 Git Bash 命令行。

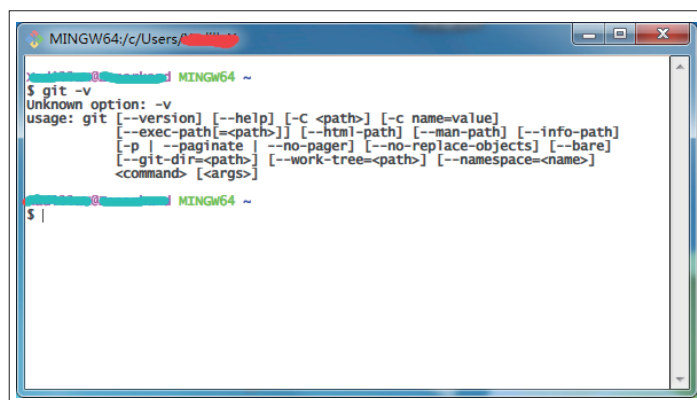


图 1-22 Windows 下的 Git Bash 命令行

安装完之后，从未使用过 Git 的读者，可以打开初始安装后出现在“开始”菜单中的 Git Bash 项打开 Git 命令行，也可以在此使用 Node.js 之类的程序，或者直接在 Windows 的命令行中使用 `git` 命令。使用者可根据自己的个人习惯决定如何使用。此前在 Windows 中就会使用 Git 的读者按照自己以前的习惯使用即可。图 1-23 为在 Windows 下 CMD 中的 Git 命令行。

¹ Git 是一款免费、开源的分布式版本控制系统，用于敏捷、高效地处理任何或小或大的项目。

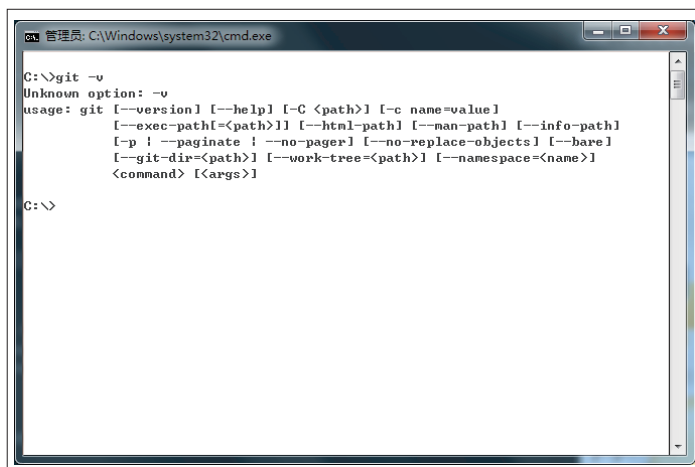


图 1-23 CMD 窗口中的 Git 命令行

• 其他问题

如果在 Windows 系统中安装 node-gyp 的前驱依赖时遇到一些问题，可以访问 <https://github.com/Microsoft/nodejs-guidelines/blob/master/windows-environment.md#compiling-native-addon-modules> 了解更多信息。

C++ 11

这里有比较重要的一点，即从 io.js 3.0 开始以及 Node.js 4 之后的版本，都需要使用 C++ 11 来对插件进行编译。所以，基本上现如今的 Node.js 下 C++ 扩展开发和安装，都需要让计算机（或者服务器）上的 C++ 编译器支持 C++ 11 了。

2. 安装

node-gyp 的安装非常简单。由于它是 Node.js 的一个常规包，因此可以直接通过 NPM 进行管理。跟平常的 Node.js 包一样，node-gyp 只需要通过 `npm install` 就能安装了。

不过需要注意的是，node-gyp 需要被全局安装，才能在安装一些 Node.js 下的 C++ 扩展时被用到，也就是说在 `npm install` 时需要加上 `-g` 的参数。

```
$ [sudo] npm install -g node-gyp

/Workspace/.npm/versions/node/v6.9.4/bin/node-gyp -> /Workspace/.npm/
versions/node/v6.9.4/lib/node_modules/node-gyp/bin/node-gyp.js
node-gyp@3.6.0 /Workspace/.npm/versions/node/v6.9.4/lib/node_modules/node-
gyp
└── rimraf@2.6.1
```

```

├── graceful-fs@4.1.11
├── semver@5.3.0
├── osenv@0.1.4 (os-tmpdir@1.0.2, os-homedir@1.0.2)
├── which@1.2.14 (isexe@2.0.0)
├── fstream@1.0.11 (inherits@2.0.3)
├── nopt@3.0.6 (abbrev@1.1.0)
├── tar@2.2.1 (inherits@2.0.3, block-stream@0.0.9)
├── minimatch@3.0.3 (brace-expansion@1.1.6)
├── mkdirp@0.5.1 (minimist@0.0.8)
├── glob@7.1.1 (path-is-absolute@1.0.1, inherits@2.0.3, fs.realpath@1.0.0,
once@1.4.0, inflight@1.0.6)
├── npmlog@4.0.2 (console-control-strings@1.1.0, set-blocking@2.0.0,
gauge@2.7.3, are-we-there-yet@1.1.2)
└── request@2.81.0 (aws-sign2@0.6.0, forever-agent@0.6.1, oauth-
sign@0.8.2, is-typedarray@1.0.0, caseless@0.12.0, tunnel-agent@0.6.0,
stringstream@0.0.5, safe-buffer@5.0.1, aws4@1.6.0, isstream@0.1.2, json-
stringify-safe@5.0.1, extend@3.0.0, performance-now@0.2.0, uuid@3.0.1,
qs@6.4.0, form-data@2.1.2, combined-stream@1.0.5, tough-cookie@2.3.2, mime-
types@2.1.15, hawk@3.1.3, http-signature@1.1.1, har-validator@4.2.1)

```

另外需要注意，本书中如果没有特殊提示，所有命令行都是在类 UNIX 系统下执行的，在 Windows 系统中可能会有略微的出入，但出入不大。其中 [sudo] 表示有可能需要以 sudo 的形式执行该命令。\$ 表示输入提示符，在 Windows 下这类似于 D:\Workspace>。如果某一行没有该输入提示符带头，就说明该行是命令行可能的输出结果。

3. node-gyp 命令

为了演示，推荐读者使用本书对应的代码“1. first build”目录下的代码。其代码可以从 GitHub 上克隆。

首先执行一遍 `node-gyp -h`，看一下它的建议帮助文档。在本书编写之际当前 `node-gyp` 的版本为 3.6.0。

```
$ node-gyp -h
```

```
Usage: node-gyp <command> [options]
```

```
where <command> is one of:
```

- build - Invokes `msbuild` and builds the module
- clean - Removes any generated build files and the "out" dir
- configure - Generates MSVC project files for the current module
- rebuild - Runs "clean", "configure" and "build" all at once
- install - Install node development files for the specified node

```
version.
- list - Prints a listing of the currently installed node development
files
- remove - Removes the node development files for the specified version
```

从上面的简易文档中可以发现，**node-gyp** 有下面几个子命令。

- **node-gyp build**: 调用 **msbuild** 以构建模块（在 **Windows** 下是 **msbuild**，若在 **UNIX** 下则会是 **make**）。
- **node-gyp clean**: 清理生成的构建文件以及 **out** 目录。
- **node-gyp configure**: 为当前模块生成 **MSVC** 项目配置文件（在 **Windows** 下是 **MSVC**，若在 **UNIX** 下则会是 **Makefile**）。
- **node-gyp rebuild**: 一次性依次执行 **clean/configure** 和 **build**。
- **node-gyp install**: 为指定版本的 **Node.js** 安装开发环境的文件。
- **node-gyp list**: 输出当前安装的 **Node.js** 开发环境文件。
- **node-gyp remove**: 移除指定版本的 **Node.js** 开发环境文件。

（1）node-gyp install

为了构建方便，我们先在当前开发环境中执行一遍 **install** 命令，将 **Node.js** 源码的一些头文件下载到本地目录。

```
$ node-gyp install
gyp info it worked if it ends with ok
gyp info using node-gyp@3.6.0
gyp info using node@6.9.4 | linux | x64
gyp WARN download NVM_NODEJS_ORG_MIRROR is deprecated and will be removed in
node-gyp v4, please use NODEJS_ORG_MIRROR
gyp http GET https://nodejs.org/dist/v6.9.4/node-v6.9.4-headers.tar.gz
gyp http 200 https://nodejs.org/dist/v6.9.4/node-v6.9.4-headers.tar.gz
gyp http GET https://nodejs.org/dist/v6.9.4/SUMS256.txt
gyp http 200 https://nodejs.org/dist/v6.9.4/SUMS256.txt
6.9.4
gyp info ok
```

若在 **Windows** 下，上述命令会把当前执行的 **Node.js** 相应版本的源码头文件、**lib** 文件等下载解压到当前用户目录的 **.node-gyp** 目录下，如 **C:\Users\USER\.node-gyp**；如果是 **Linux** 或者 **macOS** 的用户，则一样会下载解压到当前用户目录下，如 **/home/USER/.node-gyp** 目录。

我们会发现下面有相应的以版本号名为名的目录，里面就是相关的文件了。图 1-24 为 **Windows** 下 **.node-gyp** 的目录结构示例；图 1-25 为 **macOS** 下该目录结构的示例。

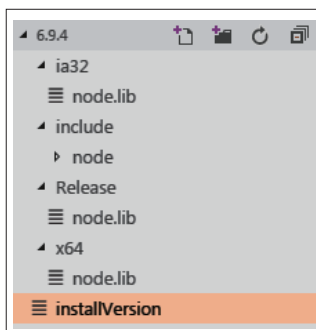


图 1-24 Windows 下的 .node-gyp 目录结构



图 1-25 macOS 下的 .node-gyp 目录

扩展阅读：install 命令有一个好玩的环境变量，叫 NODEJS_ORG_MIRROR。由于在国内访问 Node.js 官方镜像的一些文件会非常慢，因此笔者推荐使用阿里巴巴维护的 Node.js 镜像。这个时候就可以设置 NODEJS_ORG_MIRROR 这个环境变量，在设置之后当我们执行 node-gyp install 时，就已经使用了该镜像。

阿里巴巴镜像的地址根目录如下：<https://npm.taobao.org/mirrors/node>。

在 Windows 下，该目录除了头文件以外还有库文件 *.lib 等，而在 UNIX 下只有头文件。有了这些文件，才能在构建代码的时候将它们包含进项目中。

• Windows

如果是 Windows 7 用户，笔者推荐右击“电脑”，选择“属性”一项，之后依次点击“高级系统设置”→“高级”→“环境变量”，添加用户环境变量或者系统环境变量。其变量名为 NODEJS_ORG_MIRROR，其变量值为 <https://npm.taobao.org/mirrors/node>，如图 1-26 所示。



图 1-26 Windows 下的 NODEJS_ORG_MIRROR 环境变量设置

• UNIX

如果是 Linux 或者 macOS 的用户，则可以在执行命令前加上 NODEJS_ORG_MIRROR=<https://npm.taobao.org/mirrors/node> 临时修改环境变量，如：

```
$ NODEJS_ORG_MIRROR=https://npm.taobao.org/mirrors/node node-gyp install
```

(2) node-gyp configure

在构建源码之前，必须先生成项目文件。在 Windows 下是 Visual Studio 所用的 C++ 项目文件，而在 UNIX 系统下则是一个 Makefile 文件。

用命令行进入对应代码的“1. first build”目录，执行下面的命令：

```
$ node-gyp configure
```

为了使 Windows 开发者有一个更清晰的了解, 这里给出在 Windows 下执行上述命令后可能出现的输出:

```
F:\1. first build>node-gyp configure
gyp info it worked if it ends with ok
gyp info using node-gyp@3.6.0
gyp info using node@6.9.4 | win32 | x64
gyp info spawn E:\Python27\python.EXE
gyp info spawn args [ 'C:\\Users\\USER\\AppData\\Roaming\\npm\\node_
modules\\node-gyp\\gyp\\gyp_main.py',
gyp info spawn args   'binding.gyp',
gyp info spawn args   '-f',
gyp info spawn args   'msvs',
gyp info spawn args   '-G',
gyp info spawn args   'msvs_version=auto',
gyp info spawn args   '-I',
gyp info spawn args   'F:\\1. first build\\build\\config.gypi',
gyp info spawn args   '-I',
gyp info spawn args   'C:\\Users\\USER\\AppData\\Roaming\\npm\\node_
modules\\node-gyp\\addon.gypi',
gyp info spawn args   '-I',
gyp info spawn args   'C:\\Users\\USER\\.node-gyp\\6.9.4\\include\\node\\
common.gypi',
gyp info spawn args   '-Dlibrary=shared_library',
gyp info spawn args   '-Dvisibility=default',
gyp info spawn args   '-Dnode_root_dir=C:\\Users\\USER\\.node-gyp\\6.9.4',
gyp info spawn args   '-Dnode_gyp_dir=C:\\Users\\USER\\AppData\\Roaming\\
npm\\node_modules\\node-gyp',
gyp info spawn args   '-Dnode_lib_file=node.lib',
gyp info spawn args   '-Dmodule_root_dir=F:\\1. first build',
gyp info spawn args   '-Dnode_engine=v8',
gyp info spawn args   '--depth=.',
gyp info spawn args   '--no-parallel',
gyp info spawn args   '--generator-output',
gyp info spawn args   'F:\\1. first build\\build',
gyp info spawn args   '-Goutput_dir=.' ]
gyp info ok
```

这个时候我们就能看到当前目录下多了一个 build 目录。在 UNIX 环境下生成的是 Makefile 文件和一些必要的配置文件; 而在 Windows 下则是生成一个项目文件, 里面有一个 binding.sln 文件可以直接用 Visual Studio 打开, 如图 1-27 所示。

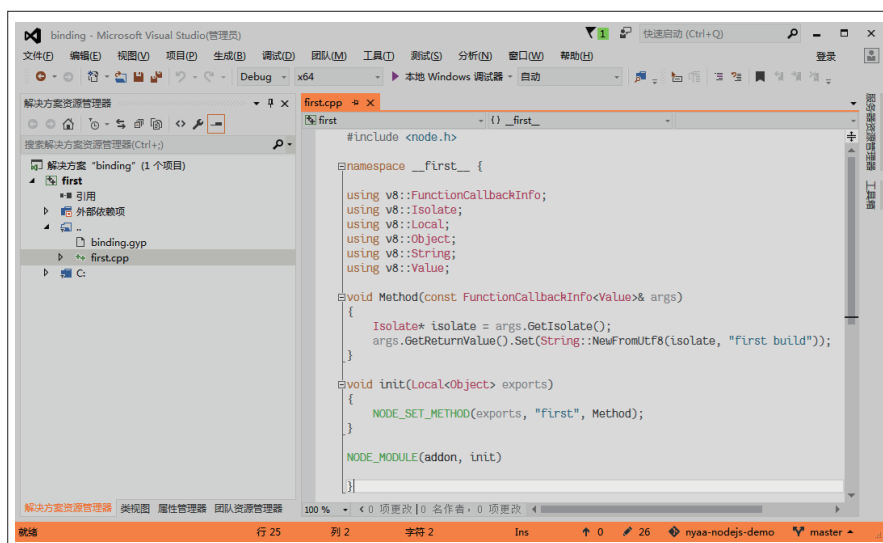


图 1-27 用 Visual Studio 打开 binding.sln 的效果

(3) node-gyp build

build 命令用于将当前所在目录的模块进行构建，将 C++ 代码编译成二进制文件。我们可以在对应代码的“1. first build”目录下尝试一下该命令。

```
$ node-gyp build
```

为了正确地编译，请使用相应的 Node.js 版本。这里可以上下偏差一些版本号，但不保证差别很大的版本能互相兼容。“1. first build”目录相应的 Node.js 版本号为 6.9.4。

为了使 Windows 开发者有一个更清晰的了解，笔者这里给出 Windows 下执行上述命令后可能出现的输出：

```
F:\1. first build>node-gyp build
gyp info it worked if it ends with ok
gyp info using node-gyp@3.6.0
gyp info using node@6.9.4 | win32 | x64
gyp info spawn C:\Program Files (x86)\MSBuild\14.0\bin\msbuild.exe
gyp info spawn args [ 'build/binding.sln',
gyp info spawn args   '/clp:Verbosity=minimal',
gyp info spawn args   '/nologo',
gyp info spawn args   '/p:Configuration=Release;Platform=x64' ]
在此解决方案中一次生成一个项目。若要启用并行生成，请添加“/m”开关。
win_delay_load_hook.cc
正在创建库 F:\1. first build\build\Release\first.lib 和对象
F:\1. first build\build\Release\first.exp
```

```
正在生成代码
已完成代码的生成
first.vcxproj -> F:\1. first build\build\Release\first.node
first.vcxproj -> F:\1. first build\build\Release\first.pdb (
Full PDB)
gyp info ok
```

我们发现执行了上述命令后，最后会在 build/Release/ 目录下生成一个 first.node。这就是我们构建的第一个 Node.js 的 C++ 模块了。

为了看一下效果，我们可以在当前目录下打开 Node.js 命令行，并引入它执行 Node.js 代码：

```
F:\1. first build>node
> const first = require("./build/Release/first")
undefined
> first.first()
'first build'
```

看到了吧，成功了！

注意：要使用 node-gyp build 命令必须先生成构建配置，即执行 node-gyp configure。

(4) node-gyp clean

clean 命令清理生成的构建文件以及 out 目录，说白了就是将目录清干净，但是源码没动。

```
$ node-gyp clean
gyp info it worked if it ends with ok
gyp info using node-gyp@3.6.0
gyp info using node@6.9.4 | darwin | x64
gyp info ok
```

(5) node-gyp rebuild

这是一个懒人命令。它一次性依次执行了 clean / configure 和 build。用通俗的语言来表达，就是先清理之前的构建记录，然后重新生成一份构建项目文件，最后进行构建编译。

```
$ node-gyp rebuild
gyp info it worked if it ends with ok
gyp info using node-gyp@3.6.0
gyp info using node@6.9.4 | darwin | x64
gyp info spawn /usr/local/bin/python2
gyp info spawn args [ '/Users/user/.nvm/versions/node/v6.9.4/lib/node_modules/node-gyp/gyp/gyp_main.py',
gyp info spawn args 'binding.gyp',
```

```

gyp info spawn args '-f',
gyp info spawn args 'make',
gyp info spawn args '-I',
gyp info spawn args '/Users/user/1. first build/build/config.gypi',
gyp info spawn args '-I',
gyp info spawn args '/Users/user/.npm/versions/node/v6.9.1/lib/node_
modules/node-gyp/addon.gypi',
gyp info spawn args '-I',
gyp info spawn args '/Users/user/.node-gyp/6.9.4/include/node/common.
gypi',
gyp info spawn args '-Dlibrary=shared_library',
gyp info spawn args '-Dvisibility=default',
gyp info spawn args '-Dnode_root_dir=/Users/user/.node-gyp/6.9.4',
gyp info spawn args '-Dnode_gyp_dir=/Users/user/.npm/versions/node/v6.9.4/
lib/node_modules/node-gyp',
gyp info spawn args '-Dnode_lib_file=node.lib',
gyp info spawn args '-Dmodule_root_dir=/Users/user/1. first build',
gyp info spawn args '-Dnode_engine=v8',
gyp info spawn args '--depth=.',
gyp info spawn args '--no-parallel',
gyp info spawn args '--generator-output',
gyp info spawn args 'build',
gyp info spawn args '-Goutput_dir=.' ]
gyp info spawn make
gyp info spawn args [ 'BUILDTYPE=Release', '-C', 'build' ]
  CXX(target) Release/obj.target/first/first.o
  SOLINK_MODULE(target) Release/first.node
gyp info ok

```

(6) 其他子命令

除了上述的子命令之外，还有就是 `list` 和 `remove` 了。

其中，`node-gyp list` 会列出以版本为维度的当前你已安装的 Node.js 开发环境头文件。

```

$ node-gyp list
gyp info it worked if it ends with ok
gyp info using node-gyp@3.6.0
gyp info using node@6.9.1 | darwin | x64
0.10.38
0.12.2
0.12.7
2.5.0
3.0.0
4.0.0
4.1.1

```

```
4.2.1
6.9.4
gyp info ok
```

上面的这个样例输出就说明了笔者的计算机中安装了 0.10.38、0.12.2、0.12.7 以及 6.9.4 等版本的 Node.js 开发环境头文件。

而 `node-gyp remove` 则是移除其中一个版本。比如笔者要移除 0.10.38，就可以执行 `node-gyp remove 0.10.38` 了。

```
$ node-gyp remove 0.10.38
gyp info it worked if it ends with ok
gyp info using node-gyp@3.6.0
gyp info using node@6.9.4 | darwin | x64
gyp info ok
```

执行完这些之后我们再执行 `node-gyp list`，就会发现 0.10.38 已经被除名了。

```
$ node-gyp list
gyp info it worked if it ends with ok
gyp info using node-gyp@3.6.0
gyp info using node@6.9.1 | darwin | x64
0.12.2
0.12.7
2.5.0
3.0.0
4.0.0
4.1.1
4.2.1
6.9.4
gyp info ok
```

除此之外，还有以下两个参数供大家使用。

- `node-gyp -h` 或者 `node-gyp --help`: 输出帮助文字。
- `node-gyp -v` 或者 `node-gyp --version`: 输出当前 `node-gyp` 的版本号。

1.4.3 其他构建工具

本小节作为扩展阅读，并不在书中详细展开，有兴趣的读者可以到其各自工具的官网查询文档。

大千世界，物种繁多。天地不仁以万物为刍狗，码农不仁以各种构建工具为刍狗。除了最受欢迎（或者说受众最广）的 `node-gyp` 以外，Node.js 中其实还有其他各种不同的构建工具。

`node-gyp` 的原理就是在安装 C++ 原生模块的时候，使用 GYP 通过一个 `binding.gyp` 文件来构建各系统所需的 C++ 项目文件（如 UNIX 下是 Makefile，Windows 下是 Visual Studio 项目），然后再通过相应的构建工具（如 Visual Studio）来进行构建。所以依照其原理，其他的构建工具大同小异，只不过 `node-gyp` 是官方支持，所以不需要嵌入任何脚本，而其他构建工具大多需要在 `package.json` 中的 `scripts` 字段加入诸如 `postinstall`¹ 的脚本，使其在包装阶段能够执行构建。

Node.js 的 C++ 模块构建工具其实有很多，读者甚至可以自己实现一套。下面列出两个构建工具供大家参考。

1. cmake-js

顾名思义，这个构建工具会使用 CMake² 当前的 C++ 源码生成一个项目文件。所以其本质上跟 `node-gyp` 一样，只是生成的项目文件（或者 Makefile）会有所不同。

它对于开发者（以及安装这个包的用户）是有要求的，需要其在计算机上安装 CMake。Windows 用户一样需要 Visual C++ Build Tools，UNIX 用户还需要 GCC。不过 UNIX 用户还多了一个选择，就是 Ninja³。

有兴趣的读者可以访问其官网进行更进一步的了解，它的 GitHub 地址是 <https://github.com/cmake-js/cmake-js>。

要使用 `cmake-js`，则需要以全局依赖的方式安装它，并且配置项目 `package.json` 文件的 `postinstall` 字段，使其在安装阶段执行 `cmake-js` 以达到构建效果。

2. node-cmake

`node-cmake` 是另一个以 CMake 为基础的构建工具。其原理与前面的 `cmake-js` 差不多，都是通过 CMakeList.txt 生成项目所需的文件，然后再通过 `postinstall` 字段进行构建和编译的。

有兴趣的读者可以访问其官方网站进行深入的了解，它的 GitHub 地址是 <https://github.com/cjntaylor/node-cmake>。

1 `scripts` 字段在 1.2.1 节中曾介绍过，而其中的 `postinstall` 在 Node.js 中表示 NPM 在安装完当前包时执行的脚本，更多相关内容可以查看 NPM 官方文档“node-scripts”一章：<https://docs.npmjs.com/misc/scripts>。

2 CMake 是一个跨平台的安装（编译）工具，可以用简单的语句来描述所有平台的安装（编译过程）。它能够输出各种各样的 Makefile 或者项目文件。

3 这里的 Ninja 是指 Ninja Build System，它是一个比较高效的构建工具。如谷歌的 Chrome 浏览器就是使用 Ninja 进行管理构建的，更多 Ninja 的内容可以访问其官网进行了解：<https://ninja-build.org/>。

1.4.4 小结

本节主要介绍了作为一个 Node.js 的 C++ 扩展开发者所需要事先搭建好的开发环境。毕竟编辑器 / IDE 作为开发者的主要装备，还是有必要好好挑选和使用的，这也就是所谓的“磨刀不误砍柴工”。而 node-gyp 作为 Node.js 界中最流行的原生扩展构建工具，也作为本书中所用的原生扩展构建工具，在阅读本书前也是需要事先安装好的。

除此之外，本节还拓展阅读了其他的一些非 GYP 系的构建工具，如 cmake-js 和 node-cmake。大家需要清楚，构建工具的本质就是通过一个配置文件和构建工具的程序本身来生成一个关于当前项目的 Makefile 或者 Visual Studio 项目文件，从而可以调用其相应的编译器对项目进行编译。

1.4.5 参考资料

- [1] Neovim: <https://neovim.io/>.
- [2] (美) 罗宾斯, (美) 汉娜, (美) 拉姆. 学习 vi 和 Vim 编辑器 [M]. 南京: 东南大学出版社, 2011.
- [3] Emacs 和 Vim: 神的编辑器和编辑器之神: <http://os.51cto.com/art/201101/242518.htm>.
- [4] Sublime Text: 学习资源篇: <http://www.jianshu.com/p/d1b9a64e2e37>.
- [5] Visual Studio Code Guide: <http://i5ting.github.io/vsc/>.
- [6] Atom 背后的故事: <https://jysperm.me/2016/11/behind-atom/>.
- [7] Visual Studio for Mac: <https://www.visualstudio.com/zh-hans/vs/visual-studio-mac/>.
- [8] 针对 Visual Studio 的 Node.js 工具: <https://www.visualstudio.com/zh-hans/vs/node-js/>.
- [9] 蒋鑫. Git 权威指南 [M]. 北京: 机械工业出版社, 2011.
- [10] node-gyp/README.md at master: <https://github.com/nodejs/node-gyp/blob/master/README.md>.

2

C++ 模块原理简析

第 1 章介绍了开发者在开发 C++ 模块时所需要具有的知识储备，本章会就 C++ 模块的一些原理进行简析。

2.1 为什么要写 C++ 模块

在写 C++ 模块之前，我们有必要问自己一个问题：为什么要写 C++ 模块？用 JavaScript 原生模块不行吗？

这里笔者大致从两个方面来回答这个问题。

2.1.1 C++ 比 JavaScript 解释器高效

其实笔者想了很久也没太想明白怎么给本小节取节名。因为市面上大多数 JavaScript 解释器就是用 C++ 写的，所以这个标题也不是特别稳妥。

笔者对此做一个补充，就是相同意思的代码，在 JavaScript 解释器中执行 JavaScript 代码的效率通常比直接执行一个 C++ 编译好后的二进制文件要低。

当然这不是绝对的，本节特指那些非并行、计算密集型的代码。因为模型的不同，单线程下实现 C++ 的 Web 请求处理和有着异步 I/O 优势的 Node.js 下实现的 Web 请求处理也是不一定能相提并论的——Node.js 底层偷偷用了别的线程。

1. NBody

这里从 *The Computer Language Benchmarks Game* (简称 CLBG) 中摘一段 Node.js 和 C++ 对于同一个问题编码执行之后的性能对比。题目就是 NBody¹。以下几段代码为遵循 BSD 协议的开源代码，在此附上开源协议链接：<https://benchmarksgame.alioth.debian.org/license.html>。

(1) 问题背景

不同语言使用相同的简易辛积分器²来模拟类木行星的轨道。

在 CLBG 里面对于该问题给出了 Java 版的样例辛积分器代码，供其他语言临摹。

```
/* The Computer Language Benchmarks Game
   http://benchmarksgame.alioth.debian.org/

   contributed by Mark C. Lewis
   modified slightly by Chad Whipkey
*/

public final class nbody {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);

        NBodySystem bodies = new NBodySystem();
        System.out.printf("%.9f\n", bodies.energy());
        for (int i=0; i<n; ++i)
            bodies.advance(0.01);
        System.out.printf("%.9f\n", bodies.energy());
    }
}

final class NBodySystem {
    private Body[] bodies;

    public NBodySystem(){
        bodies = new Body[]{
            Body.sun(),
            Body.jupiter(),
            Body.saturn(),
            Body.uranus(),
            Body.neptune()
        };
    }
}
```

1 该题目的地址：<https://benchmarksgame.alioth.debian.org/u64q/nbody-description.html>。

2 在太阳系动力学中，辛积分器是研究哈密顿系统的长期定性演化的最佳工具。在本书中该概念仅为题目引申，无须过于纠结。

```

double px = 0.0;
double py = 0.0;
double pz = 0.0;
for(int i=0; i < bodies.length; ++i) {
    px += bodies[i].vx * bodies[i].mass;
    py += bodies[i].vy * bodies[i].mass;
    pz += bodies[i].vz * bodies[i].mass;
}
bodies[0].offsetMomentum(px,py,pz);
}

public void advance(double dt) {
    for(int i=0; i < bodies.length; ++i) {
        Body iBody = bodies[i];
        for(int j=i+1; j < bodies.length; ++j) {
            double dx = iBody.x - bodies[j].x;
            double dy = iBody.y - bodies[j].y;
            double dz = iBody.z - bodies[j].z;

            double dSquared = dx * dx + dy * dy + dz * dz;
            double distance = Math.sqrt(dSquared);
            double mag = dt / (dSquared * distance);

            iBody.vx -= dx * bodies[j].mass * mag;
            iBody.vy -= dy * bodies[j].mass * mag;
            iBody.vz -= dz * bodies[j].mass * mag;

            bodies[j].vx += dx * iBody.mass * mag;
            bodies[j].vy += dy * iBody.mass * mag;
            bodies[j].vz += dz * iBody.mass * mag;
        }
    }

    for ( Body body : bodies) {
        body.x += dt * body.vx;
        body.y += dt * body.vy;
        body.z += dt * body.vz;
    }
}

public double energy(){
    double dx, dy, dz, distance;
    double e = 0.0;

    for (int i=0; i < bodies.length; ++i) {

```

```

        Body iBody = bodies[i];
        e += 0.5 * iBody.mass *
            ( iBody.vx * iBody.vx
              + iBody.vy * iBody.vy
              + iBody.vz * iBody.vz );

        for (int j=i+1; j < bodies.length; ++j) {
            Body jBody = bodies[j];
            dx = iBody.x - jBody.x;
            dy = iBody.y - jBody.y;
            dz = iBody.z - jBody.z;

            distance = Math.sqrt(dx*dx + dy*dy + dz*dz);
            e -= (iBody.mass * jBody.mass) / distance;
        }
    }
    return e;
}

final class Body {
    static final double PI = 3.141592653589793;
    static final double SOLAR_MASS = 4 * PI * PI;
    static final double DAYS_PER_YEAR = 365.24;

    public double x, y, z, vx, vy, vz, mass;

    public Body(){}

    static Body jupiter(){
        Body p = new Body();
        p.x = 4.84143144246472090e+00;
        p.y = -1.16032004402742839e+00;
        p.z = -1.03622044471123109e-01;
        p.vx = 1.66007664274403694e-03 * DAYS_PER_YEAR;
        p.vy = 7.69901118419740425e-03 * DAYS_PER_YEAR;
        p.vz = -6.90460016972063023e-05 * DAYS_PER_YEAR;
        p.mass = 9.54791938424326609e-04 * SOLAR_MASS;
        return p;
    }

    static Body saturn(){
        Body p = new Body();
        p.x = 8.34336671824457987e+00;
        p.y = 4.12479856412430479e+00;
        p.z = -4.03523417114321381e-01;
    }
}

```

```

    p.vx = -2.76742510726862411e-03 * DAYS_PER_YEAR;
    p.vy = 4.99852801234917238e-03 * DAYS_PER_YEAR;
    p.vz = 2.30417297573763929e-05 * DAYS_PER_YEAR;
    p.mass = 2.85885980666130812e-04 * SOLAR_MASS;
    return p;
}

static Body uranus(){
    Body p = new Body();
    p.x = 1.28943695621391310e+01;
    p.y = -1.51111514016986312e+01;
    p.z = -2.23307578892655734e-01;
    p.vx = 2.96460137564761618e-03 * DAYS_PER_YEAR;
    p.vy = 2.37847173959480950e-03 * DAYS_PER_YEAR;
    p.vz = -2.96589568540237556e-05 * DAYS_PER_YEAR;
    p.mass = 4.36624404335156298e-05 * SOLAR_MASS;
    return p;
}

static Body neptune(){
    Body p = new Body();
    p.x = 1.53796971148509165e+01;
    p.y = -2.59193146099879641e+01;
    p.z = 1.79258772950371181e-01;
    p.vx = 2.68067772490389322e-03 * DAYS_PER_YEAR;
    p.vy = 1.62824170038242295e-03 * DAYS_PER_YEAR;
    p.vz = -9.51592254519715870e-05 * DAYS_PER_YEAR;
    p.mass = 5.15138902046611451e-05 * SOLAR_MASS;
    return p;
}

static Body sun(){
    Body p = new Body();
    p.mass = SOLAR_MASS;
    return p;
}

Body offsetMomentum(double px, double py, double pz){
    vx = -px / SOLAR_MASS;
    vy = -py / SOLAR_MASS;
    vz = -pz / SOLAR_MASS;
    return this;
}
}

```

该问题将测试当 N 为 50 000 000 情况下程序的执行时间。

(2) 摘录一份 C++ 代码

```
/* The Computer Language Benchmarks Game
   http://benchmarksgame.alioth.debian.org/

   contributed by Mark C. Lewis
   modified slightly by Chad Whipkey
   converted from java to c++, added sse support, by Branimir Maksimovic
   modified by Vaclav Zeman
*/

#include <cstdio>
#include <cmath>
#include <cstdlib>
#include <array>
#include <immintrin.h>

static const double PI = 3.141592653589793;
static const double SOLAR_MASS = 4 * PI * PI;
static const double DAYS_PER_YEAR = 365.24;

class Body {
public:
    double x, y, z, filler, vx, vy, vz, mass;

    Body() {}

    static Body& jupiter() {
        static Body p;
        p.x = 4.84143144246472090e+00;
        p.y = -1.16032004402742839e+00;
        p.z = -1.03622044471123109e-01;
        p.vx = 1.66007664274403694e-03 * DAYS_PER_YEAR;
        p.vy = 7.69901118419740425e-03 * DAYS_PER_YEAR;
        p.vz = -6.90460016972063023e-05 * DAYS_PER_YEAR;
        p.mass = 9.54791938424326609e-04 * SOLAR_MASS;
        return p;
    }

    static Body& saturn() {
        static Body p;
        p.x = 8.34336671824457987e+00;
```

```

    p.y = 4.12479856412430479e+00;
    p.z = -4.03523417114321381e-01;
    p.vx = -2.76742510726862411e-03 * DAYS_PER_YEAR;
    p.vy = 4.99852801234917238e-03 * DAYS_PER_YEAR;
    p.vz = 2.30417297573763929e-05 * DAYS_PER_YEAR;
    p.mass = 2.85885980666130812e-04 * SOLAR_MASS;
    return p;
}

static Body& uranus(){
    static Body p;
    p.x = 1.28943695621391310e+01;
    p.y = -1.51111514016986312e+01;
    p.z = -2.23307578892655734e-01;
    p.vx = 2.96460137564761618e-03 * DAYS_PER_YEAR;
    p.vy = 2.37847173959480950e-03 * DAYS_PER_YEAR;
    p.vz = -2.96589568540237556e-05 * DAYS_PER_YEAR;
    p.mass = 4.36624404335156298e-05 * SOLAR_MASS;
    return p;
}

static Body& neptune(){
    static Body p;
    p.x = 1.53796971148509165e+01;
    p.y = -2.59193146099879641e+01;
    p.z = 1.79258772950371181e-01;
    p.vx = 2.68067772490389322e-03 * DAYS_PER_YEAR;
    p.vy = 1.62824170038242295e-03 * DAYS_PER_YEAR;
    p.vz = -9.51592254519715870e-05 * DAYS_PER_YEAR;
    p.mass = 5.15138902046611451e-05 * SOLAR_MASS;
    return p;
}

static Body& sun(){
    static Body p;
    p.mass = SOLAR_MASS;
    return p;
}

Body& offsetMomentum(double px, double py, double pz){
    vx = -px / SOLAR_MASS;
    vy = -py / SOLAR_MASS;
    vz = -pz / SOLAR_MASS;
    return *this;
}

};

```

```

class NBodySystem {
private:
    std::array<Body, 5> bodies;

public:
    NBodySystem()
        : bodies {{
            Body::sun(),
            Body::jupiter(),
            Body::saturn(),
            Body::uranus(),
            Body::neptune()
        }}
    {
        double px = 0.0;
        double py = 0.0;
        double pz = 0.0;
        for(unsigned i=0; i < bodies.size(); ++i) {
            px += bodies[i].vx * bodies[i].mass;
            py += bodies[i].vy * bodies[i].mass;
            pz += bodies[i].vz * bodies[i].mass;
        }
        bodies[0].offsetMomentum(px,py,pz);
    }

    void advance(double dt) {
        const unsigned N = (bodies.size()-1)*bodies.size()/2;
        struct __attribute__((aligned(16))) R {
            double dx,dy,dz,filler;
        };
        static R r[1000];
        static __attribute__((aligned(16))) double mag[1000];

        for(unsigned i=0,k=0; i < bodies.size()-1; ++i) {
            Body& iBody = bodies[i];
            for(unsigned j=i+1; j < bodies.size(); ++j,++k) {
                r[k].dx = iBody.x - bodies[j].x;
                r[k].dy = iBody.y - bodies[j].y;
                r[k].dz = iBody.z - bodies[j].z;
            }
        }

        for(unsigned i=0; i < N; i+=2) {
            __m128d dx,dy,dz;

```



```

dx = _mm_loadl_pd(dx, &r[i].dx);
dy = _mm_loadl_pd(dy, &r[i].dy);
dz = _mm_loadl_pd(dz, &r[i].dz);

dx = _mm_loadh_pd(dx, &r[i+1].dx);
dy = _mm_loadh_pd(dy, &r[i+1].dy);
dz = _mm_loadh_pd(dz, &r[i+1].dz);

__m128d dSquared = dx*dx + dy*dy + dz*dz;

__m128d distance =
    _mm_cvtps_pd(_mm_rsqrt_ps(_mm_cvtpd_ps(dSquared)));
for(unsigned j=0; j<2; ++j)
{
    distance = distance * _mm_set1_pd(1.5) -
        ((_mm_set1_pd(0.5) * dSquared) * distance) *
        (distance * distance);
}

__m128d dmag = _mm_set1_pd(dt)/(dSquared) * distance;
__mm_store_pd(&mag[i], dmag);
}

for(unsigned i=0, k=0; i < bodies.size()-1; ++i) {
    Body& iBody = bodies[i];
    for(unsigned j=i+1; j < bodies.size(); ++j, ++k) {
        iBody.vx -= r[k].dx * bodies[j].mass * mag[k];
        iBody.vy -= r[k].dy * bodies[j].mass * mag[k];
        iBody.vz -= r[k].dz * bodies[j].mass * mag[k];

        bodies[j].vx += r[k].dx * iBody.mass * mag[k];
        bodies[j].vy += r[k].dy * iBody.mass * mag[k];
        bodies[j].vz += r[k].dz * iBody.mass * mag[k];
    }
}

for (unsigned i = 0; i < bodies.size(); ++i) {
    bodies[i].x += dt * bodies[i].vx;
    bodies[i].y += dt * bodies[i].vy;
    bodies[i].z += dt * bodies[i].vz;
}
}

double energy(){
    double e = 0.0;

```

```

        for (unsigned i=0; i < bodies.size(); ++i) {
            Body const & iBody = bodies[i];
            double dx, dy, dz, distance;
            e += 0.5 * iBody.mass *
                ( iBody.vx * iBody.vx
                  + iBody.vy * iBody.vy
                  + iBody.vz * iBody.vz );

            for (unsigned j=i+1; j < bodies.size(); ++j) {
                Body const & jBody = bodies[j];
                dx = iBody.x - jBody.x;
                dy = iBody.y - jBody.y;
                dz = iBody.z - jBody.z;

                distance = sqrt(dx*dx + dy*dy + dz*dz);
                e -= (iBody.mass * jBody.mass) / distance;
            }
        }
        return e;
    }
};

int main(int argc, char** argv) {
    int n = atoi(argv[1]);

    NBodySystem bodies;
    printf("%.9f\n", bodies.energy());
    for (int i=0; i<n; ++i)
        bodies.advance(0.01);
    printf("%.9f\n", bodies.energy());
}

```

该代码在四核 Ubuntu 下，以 G++ 6.3 进行编译，编译命令如下：

```

$ g++ -c -pipe -O3 -fomit-frame-pointer -march=native -mfpmath=sse -msse3
--std=c++11 nbody.gpp-3.c++ -o nbody.gpp-3.c++.o && /usr/bin/g++ nbody.gpp-
3.c++.o -o nbody.gpp-3.gpp_run -fopenmp

```

编译完成之后使用下面的命令执行：

```

$ ./nbody.gpp-3.gpp_run 50000000

```

程序的执行时间约为 9.67 秒。

(3) 摘录一份 Node.js 代码

```

/* The Computer Language Benchmarks Game
   http://benchmarksgame.alioth.debian.org/
   contributed by Isaac Gouy */

var PI = 3.141592653589793;
var SOLAR_MASS = 4 * PI * PI;
var DAYS_PER_YEAR = 365.24;

function Body(x,y,z,vx,vy,vz,mass){
    this.x = x;
    this.y = y;
    this.z = z;
    this.vx = vx;
    this.vy = vy;
    this.vz = vz;
    this.mass = mass;
}

Body.prototype.offsetMomentum = function(px,py,pz) {
    this.vx = -px / SOLAR_MASS;
    this.vy = -py / SOLAR_MASS;
    this.vz = -pz / SOLAR_MASS;
    return this;
}

function Jupiter(){
    return new Body(
        4.84143144246472090e+00,
        -1.16032004402742839e+00,
        -1.03622044471123109e-01,
        1.66007664274403694e-03 * DAYS_PER_YEAR,
        7.69901118419740425e-03 * DAYS_PER_YEAR,
        -6.90460016972063023e-05 * DAYS_PER_YEAR,
        9.54791938424326609e-04 * SOLAR_MASS
    );
}

function Saturn(){
    return new Body(
        8.34336671824457987e+00,
        4.12479856412430479e+00,
        -4.03523417114321381e-01,
        -2.76742510726862411e-03 * DAYS_PER_YEAR,
        4.99852801234917238e-03 * DAYS_PER_YEAR,

```

```

        2.30417297573763929e-05 * DAYS_PER_YEAR,
        2.85885980666130812e-04 * SOLAR_MASS
    );
}

function Uranus(){
    return new Body(
        1.28943695621391310e+01,
        -1.51111514016986312e+01,
        -2.23307578892655734e-01,
        2.96460137564761618e-03 * DAYS_PER_YEAR,
        2.37847173959480950e-03 * DAYS_PER_YEAR,
        -2.96589568540237556e-05 * DAYS_PER_YEAR,
        4.36624404335156298e-05 * SOLAR_MASS
    );
}

function Neptune(){
    return new Body(
        1.53796971148509165e+01,
        -2.59193146099879641e+01,
        1.79258772950371181e-01,
        2.68067772490389322e-03 * DAYS_PER_YEAR,
        1.62824170038242295e-03 * DAYS_PER_YEAR,
        -9.51592254519715870e-05 * DAYS_PER_YEAR,
        5.15138902046611451e-05 * SOLAR_MASS
    );
}

function Sun(){
    return new Body(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, SOLAR_MASS);
}

function NBodySystem(bodies){
    this.bodies = bodies;
    var px = 0.0;
    var py = 0.0;
    var pz = 0.0;
    var size = this.bodies.length;
    for (var i=0; i<size; i++){
        var b = this.bodies[i];
        var m = b.mass;
        px += b.vx * m;
        py += b.vy * m;
        pz += b.vz * m;
    }
}

```

```

    }
    this.bodies[0].offsetMomentum(px,py,pz);
}

NBodySystem.prototype.advance = function(dt){
    var dx, dy, dz, distance, mag;
    var size = this.bodies.length;

    for (var i=0; i<size; i++) {
        var bodyi = this.bodies[i];
        for (var j=i+1; j<size; j++) {
            var bodyj = this.bodies[j];
            dx = bodyi.x - bodyj.x;
            dy = bodyi.y - bodyj.y;
            dz = bodyi.z - bodyj.z;

            distance = Math.sqrt(dx*dx + dy*dy + dz*dz);
            mag = dt / (distance * distance * distance);

            bodyi.vx -= dx * bodyj.mass * mag;
            bodyi.vy -= dy * bodyj.mass * mag;
            bodyi.vz -= dz * bodyj.mass * mag;

            bodyj.vx += dx * bodyi.mass * mag;
            bodyj.vy += dy * bodyi.mass * mag;
            bodyj.vz += dz * bodyi.mass * mag;
        }
    }

    for (var i=0; i<size; i++) {
        var body = this.bodies[i];
        body.x += dt * body.vx;
        body.y += dt * body.vy;
        body.z += dt * body.vz;
    }
}

NBodySystem.prototype.energy = function(){
    var dx, dy, dz, distance;
    var e = 0.0;
    var size = this.bodies.length;

    for (var i=0; i<size; i++) {
        var bodyi = this.bodies[i];

        e += 0.5 * bodyi.mass *

```

```

        ( bodyi.vx * bodyi.vx
        + bodyi.vy * bodyi.vy
        + bodyi.vz * bodyi.vz );

    for (var j=i+1; j<size; j++) {
        var bodyj = this.bodies[j];
        dx = bodyi.x - bodyj.x;
        dy = bodyi.y - bodyj.y;
        dz = bodyi.z - bodyj.z;

        distance = Math.sqrt(dx*dx + dy*dy + dz*dz);
        e -= (bodyi.mass * bodyj.mass) / distance;
    }
}
return e;
}

var n = +process.argv[2];
var bodies = new NBodySystem( Array(
    Sun(), Jupiter(), Saturn(), Uranus(), Neptune()
));

console.log(bodies.energy().toFixed(9));
for (var i=0; i<n; i++){ bodies.advance(0.01); }
console.log(bodies.energy().toFixed(9));

```

将该 Node.js 代码放到同样的 Ubuntu 机器下，使用 7.9.0 版本的 Node.js 执行。

```
$ node nbody.js 50000000
```

其执行时间约为 27.76 秒。

(4) 对比结果

在 9.67 秒和 27.76 秒的对比下，我们很容易能发现 C++ 在该题目的对比下完胜 Node.js。实际上类似的 C++ 完胜结果在 CLBG 里面有很多，读者可以自行去 <https://benchmarksgame.alieth.debian.org/> 查看。

这时可能很多读者会问：既然 C++ 效率那么高，为什么不整个项目都用 C++ 写呢？其实这也是不科学的。C++ 的效率固然高，但其所需要的维护成本和开发效率与 Node.js 也不在一个层次上。所以我们也需要在一定程度上权衡项目的开发效率、维护成本以及性能才行。所以在一个项目中，整体使用 Node.js 来完成，偶尔在里面维护一两个使用 C++ 写的扩展，这也是一个非常奇妙的体验。

2. 正则趣事

有时候 C++ 写出来的包虽然在性能上稍稍高于 Node.js 用 JavaScript 写出来的代码，但这高出来的性能如果还抵不过 Node.js 打开并执行 C++ 扩展时所消耗掉的 I/O 时间或者在 V8 数据结构与你自行设计的数据结构之间进行转换所消耗的时间（常见的一种情况就是要将一个大的 JavaScript 对象遍历并转换为你能用的一个数据结构），这个时候用 C++ 反而有点得不偿失了。

在 CLBG 里面有一道关于正则表达式的题目，地址是 <https://benchmarksgame.alioth.debian.org/u64q/regexredux-description.html>，该题目的大意是，在不同语言下使用一些相同的简易正则表达式去操作一个从文件读取的结果。

我们发现最快的是 C 语言，为 1.45 秒；Node.js 的结果也比较靠前，是 4.15 秒；而 C++ 的结果是 17.14 秒，让人意想不到。

不过待我们冷静下来分析一下之后，发现 C 语言那版作者用的是 PCRE¹ 这个正则表达式解析库里面的正则表达式，Node.js 用的是 V8 带的正则表达式解析库 Irregexp²，而 C++ 版本的作者用的则是 boost::regex³ 的正则表达式解析库。

从一个较早期的性能对比报告来看，大部分情况下这 3 种正则表达式解析库的性能从高到低依次为 PCRE、V8 到 boost::regex，如表 2-1 所示。

表 2-1 3 种正则表达式解析库的性能对比（秒）⁴

库名	URI	Email	Date	Sum3	URI and Email
boost::regex (1.42)	5.44	1.82	1.15	8.41	9.84
PCRE-posix (8.10)	0.67	0.39	0.54	1.60	13.48
Javascript V8 (2.3.1)	2.30	2.57	1.17	6.04	3.70

这里笔者为表 2-1 的几个正则表达式做一个解释。

- ① URI: `([a-zA-Z][a-zA-Z0-9]*)://([^ /]+)(/[^]*)?`
- ② Email: `([^ @]+)@([^ @]+)`
- ③ Date: `([0-9][0-9]?)/([0-9][0-9]?)/([0-9][0-9]([0-9][0-9])?)`
- ④ URI and Email: `([a-zA-Z][a-zA-Z0-9]*)://([^ /]+)(/[^]*)?|([^ @]+)@([^ @]+)`

1 PCRE 的全称为 Perl Compatible Regular Expressions，是源自 Perl 正则表达式实现的一套 C 语言正则表达式库。大名鼎鼎的 Web 服务器 Nginx 就是使用 PCRE 的，更多信息可以访问其官网获取：<http://www.pcre.org/>。

2 Irregexp 是 V8 所使用的一套正则表达式引擎，这是 V8 在解析正则表达式时非常高效的原因，其仓库地址是 <https://github.com/ashinn/irregexp>。

3 boost 是一组扩充 C++ 功能的经过同行评审（Peer-reviewed）且开源的程序库。boost::regex 是其对正则表达式处理的实现。

4 数据来源为 *Benchmark of Regex Libraries*：<http://lh3lh3.users.sourceforge.net/reb.shtml>。

所以一份代码是否使用 C++ 更为高效只是站在一个宏观的角度进行剖析。在某些情况下或者由于实现的问题，C++ 的效率可能不一定比 Node.js 高。不过通常情况下，在一些计算密集型的代码实现上，Node.js 的 C++ 扩展的效率会比使用 JavaScript 来实现要高。

2.1.2 已有的 C++ 轮子

还有一种比较常见的使用 C++ 扩展的原因是市面上或者手头已有一套 C++ 的轮子，且用 Node.js 重新实现一遍非常麻烦或者不现实，这个时候就可以基于这个轮子包一层 C++ 的扩展了——当然前提是你的项目主体本身就是 Node.js，否则就是多此一举了。

1. 阿里云 ONS

这里说一个笔者自身遇到过的例子，那就是阿里云 ONS¹ 了。阿里巴巴内部之前就有一套从 Java 源码仿制过来的 Node.js 客户端 SDK，不过久久未向外界发布。后来阿里巴巴官方于 2014 年白色情人节 3 月 14 日发布其 ONS 的第一个版本 Node.js 客户端 SDK。不过，这不是重点。

重点是 ONS 本身的协议（也就是 RocketMQ）并没有一个很好的文档，也没有开源的 ONS 客户端源码可以参考，因为纯仿制 RocketMQ SDK 代码的话会有小部分兼容上的问题——ONS 有一些额外的内容存在。

既然其一没有协议文档，二没有开源的其他语言可仿制代码，这个时候怎么办呢？

于是笔者很机智地将目光放到了其官网上公布的 C++ SDK——它由一堆头文件、一份文档和两个平台的静态链接库组成。瞧，这不就可以开工了吗？

在官方的 Node.js 版本 ONS SDK 出来之前，笔者自己造了一个基于其官方 C++ 版本的 ONS SDK 封装的轮子，用的当然是本书所讲的姿势——Node.js 的 C++ 扩展了。

有兴趣的读者可以到该 SDK 的仓库进行围观：<https://github.com/xadillax/alipay-ons>。

而且截至笔者编写本书之际，通过对比两个库暴露出来的 API，可以看出笔者实现的 ONS SDK 中生产者有生产顺序消息的能力，而官方 SDK 在最底层虽然有预留逻辑，但是并未在使用者层面上暴露出来。这个优势得益于其 C++ 的 SDK 实现了对顺序消息的支持。

2. Bling Hashes

这里还有一个例子，与上面 ONS 的情况略不一样。

¹ 阿里云 ONS 是一个基于阿里巴巴开源的 RocketMQ 推出的互联网消息队列服务。之后其内核 RocketMQ 在 2016 年被阿里巴巴捐赠给 Apache 软件基金会。ONS 产品官网为 <https://www.aliyun.com/product/ons>。

Bling Hashes 是笔者以前为了在 Node.js 实现一套计算字符串哈希的 C++ 扩展。其算法是在 ACM 界¹中常见的哈希算法。网上有很多人都会收集这样的代码作为自己的备用代码库。

当时笔者实现的时候直接摘抄了 ByVoid² 博客上收录的一些 ACM 常用字符串哈希函数，并对其进行 C++ 的封装。ByVoid 为这些算法进行了一次评分，其结果如表 2-2 所示。

表 2-2 常用字符串哈希函数评分³

Hash 函数	数据 1	数据 2	数据 3	数据 4	数据 1 得分	数据 2 得分	数据 3 得分	数据 4 得分	平均分
BKDRHash	2	0	4774	481	96.55	100	90.95	82.05	92.64
APHash	2	3	4754	493	96.55	88.46	100	51.28	86.28
DJBHash	2	2	4975	474	96.55	92.31	0	100	83.43
JSHash	1	4	4761	506	100	84.62	96.83	17.95	81.94
RSHash	1	0	4861	505	100	100	51.58	20.51	75.96
SDBMHash	3	2	4849	504	93.1	92.31	57.01	23.08	72.41
PJWHash	30	26	4878	513	0	0	43.89	0	21.95
ELFHash	30	26	4878	513	0	0	43.89	0	21.95

表 2-2 进行评测的原则如下：

数据 1 为 100000 个字母和数字组成的随机串哈希冲突个数。数据 2 为 100000 个有意义的英文句子哈希冲突个数。数据 3 为数据 1 的哈希值与 1000003（大素数）求模后存储到线性表中冲突的个数。数据 4 为数据 1 的哈希值与 10000019（更大素数）求模后存储到线性表中冲突的个数。

经过比较，得出以上平均得分。平均数为平方平均数。

为了讲解方便，书中给出其中评测分最高的 BKDRHash 算法的代码。

```
unsigned int BKDRHash(char *str)
{
    unsigned int seed = 131; // 31 131 1313 13131 131313 etc..
    unsigned int hash = 0;

    while (*str)
    {
```

1 这里的 ACM 界是指 ACM/ICPC 程序设计大赛界，全称为 ACM 国际大学生程序设计竞赛（ACM International Collegiate Programming Contest）。

2 ByVoid 的真名为郭家宝，是当年有名的竞赛选手，并且是国内早期使用 Node.js 的人之一，著有国内早期 Node.js 教程书《Node.js 开发指南》。

3 表 2-2 以及本节所涉及的未特殊说明的字符串哈希算法均摘自 ByVoid 的个人博客：<https://www.byvoid.com/zhs/blog/string-hash-compare>。

```

        hash = hash * seed + (*str++);
    }

    return (hash & 0x7FFFFFFF);
}

```

在 C++ 中我们能比较随意地借助数据边界溢出的特性，比较方便地进行乘法操作和加法操作来截取某个边界值，并且能比较方便地使用位运算进行某位的修正。

而如果上面的代码使用 JavaScript 实现，那么我们就免不了要非常细致地了解 ECMAScript 规范以知道各符号操作后的结果，尤其是数据边界的情况（笔者相信其实很多人虽然了解 ECMAScript 的大致情况，但是要具体到某一条规范的时候仍需要翻阅 ECMAScript 规范文档，将其作为工具书来用）。而且 ECMAScript 中所有的数值都是浮点数，哪怕是执行 `parseInt` 之后在接下来的使用中还是会在底层变回一个浮点数。并且实际上执行 `parseInt` 返回的结果并不是无符号整型的，也就是说边界值也不一样。这给我们将上述代码转用 JavaScript 来实现制造了困难，哪怕不是困难，也会消耗一些脑力来考虑如何转换。

还有一个需要考虑的问题，那就是就算你转了一个函数，这里却还有许多函数等着你去转。

直接站在巨人的肩膀上使用公开的源码不好吗？一定要从头造吗？于是——使用 **Node.js** 的 C++ 扩展来封装吧，把上面的函数原封不动地复制到你的代码中并很有良心地注明出处，然后包一层扩展来完成字符串哈希包的实现吧。

什么？这样还不能说服你？那笔者要“祭”出在 **Bling Hashes** 包里面的另一种算法了——出自谷歌之手的 **CityHash**：

CityHash 是一系列的非加密哈希函数，是为了快速计算出字符串哈希，其效率在 64 位 CPU 上得到了特殊优化。在工业级应用中可以使用该哈希算法。不过其缺点是代码比同类流行算法复杂。

在这种情况下照抄一遍 **CityHash** 那么长的代码还得看不少论文，并且在转换成 JavaScript 代码时在数值边界和处理上还有非常多的“坑”，这时用它原来的代码进行 C++ 封装是再好不过了。

事实上 **Bling Hashes** 这个包里面就是这么做的。除了包含上述流行的一些字符串哈希算法之外，它还包含了 **CityHash32** 和 **CityHash64**。别问笔者为什么这里没有 **CityHash128**——JavaScript 原生能支持到这么大的数值吗？当然，实际上我们能包含两个 **Number** 的数组进行返回，因为在 **CityHash** 底层也是这么干的。

```

// CityHash128 的返回值定义
typedef std::pair<uint64, uint64> uint128;

```

最后，这里给出大家期待已久的 Bling Hashes 代码仓库地址：https://github.com/XadillaX/bling_hashes。

通过阿里云 ONS 和 Bling Hashes 的例子，我们就能很容易想象到这样的场景了：一个库要么代码太复杂，使用 JavaScript 完全实现一遍不大现实；要么其根本就没开放源码，但是恰巧都有 C++（或者 C 语言）实现的版本（无论是头文件加链接库的形式还是纯源码的形式）。这时为了完成任务或者节约开发成本，就需要使用 Node.js 的 C++ 扩展了。

3. 扩展阅读

最后这里再讲两个笔者以前写的原生扩展，造这两个轮子的理由与上面说的差不多，不过更大的原因是使用 JavaScript 难以实现——如系统底层的 API 等。

（1）Sync Runner

在早期的 Node.js 版本中，子进程的创建和执行是不支持同步的，这就导致了我们在执行某些子进程命令行时使代码结构复杂。

实际上 Node.js 异步没错，但是也要分场合。如果仅仅是为了写一个命令行的程序，在里面需要获取当前环境的 Git 版本，那么我们会想要自己能写一份这样的代码。

```
const version = child.run("git --version");
```

很遗憾，在早期的 Node.js 版本中这样的写法是不存在的，我们必须使用 Child Process 的异步 API 启动子进程，然后在回调函数中获取其结果。

当年为了在 Node.js 中使上述代码成真，笔者就写了这个 Sync Runner。虽然就现在来说已有 `childProcess.execSync` 等 API 能支持同步启动子进程了，但在之前没有该 API 的时候，这个原生扩展还是非常有用的。

有兴趣的读者可以访问 <https://github.com/xadillax/syncRunner> 获取更多相关信息。

```
// Sync Runner 使用样例
```

```
const run = require("sync-runner");
const version = run("git --version");
console.log(version);
```

（2）Real Homedir

这个包的出现源自一个环境变量出现的 Bug。

事情的起因主要是一个 Java 开发人员通过 `sudo -u admin` 来执行基于 Node.js 实现的服务挂了，然后在各种调试之后，终于找到了问题的所在。

那个 Node.js 程序会在用户路径下记录 Log 文件，而这个用户路径是由 `process.env.HOME` 获取到的。很神奇的一点是，通过它获取到的用户路径依然是原用户的路径，而不是期望中的 `/home/admin`，于是就出现了权限问题，导致写入失败。

接下来看了一下 `sudo -h` 里面对 `-u` 的解释：

```
-u user          run command (or edit file) as specified user
                  // 指定用户运行命令（或编辑文件）
```

那么，问题又来了，这个命令（`sudo -u admin`）和真正切换到该用户下（`sudo su admin`）去执行命令有什么区别呢？是不是因为这些区别才引起了一些奇怪的权限问题呢？

首先来试一下 `sudo -u xxx`。由于本地计算机上没有其他的用户，因此直接用 `root` 代替了。

```
~ sudo -u root node
process.env
{
  HOME: '/Users/minary',
  LOGNAME: 'root',
  USER: 'root',
  USERNAME: 'root',
  SUDO_COMMAND: '/Users/Minary/.nvm/versions/node/v4.2.1/bin/node',
  SUDO_USER: 'minary',
  SUDO_UID: '501',
  SUDO_GID: '20'
}
```

可以看到这里的 `HOME` 字段并不是 `/var/root` 而是 `/Users/` 当前用户，用了 `root` 的权限来执行 `node`，但是其类似 `HOME` 字段的环境变量仍未被改变，还是在当前状态。

——摘自阿里巴巴 Node.js 工程师芙兰的博客¹

后来笔者在尝试使用 `os.homedir()` 的时候发现结果也不是预期的，原因在于 `os.homedir()` 的底层实现中用的是 `libuv` 来获取用户目录，而 `libuv` 的相关方法会先去环境变量获取用户目录，这导致了其非预期的结果。当时 Node.js 的版本还没有 6，没有 `os.userInfo()` 这个 API。实际上 Node.js 6 的这个 API 也是因为笔者当时在 Node.js 的 Git 仓库提交了相关的 Issue² 才加上去的。图 2-1 就展示了在 UNIX 下 `libuv` 的 `uv_os_homedir()` 函数流程图。

1 该问题来自阿里巴巴 Node.js 研发工程师芙兰的技术博客，可访问 <http://f10.moe/2016/03/07/%E4%B8%80%E4%B8%AA%E7%94%B1%E4%BA%8E%E7%8E%AF%E5%A2%83%E5%8F%98%E9%87%8F%E4%BA%A7%E7%94%9F%E7%9A%84bug/> 获取更多详细信息。

2 有关 `os.homedir()` 的更多信息，可访问 <https://github.com/nodejs/node/issues/5582> 查看原委。

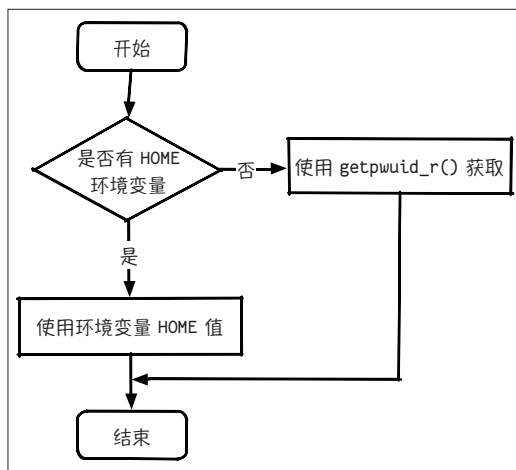


图 2-1 UNIX 下 libuv 的 uv_os_homedir() 函数流程图

在 Node.js 添加 `os.userInfo()` 之前，笔者临时用 C++ 写了原生扩展来规避这个问题。

实际上其原理很简单，就是复制了一遍 libuv 获取用户目录的方法，然后删除了该方法最开始的使用环境变量作为用户目录的一个分支条件。

有兴趣的读者可以访问 <https://github.com/BoogeeDoo/real-homedir> 获取更多相关信息。

2.1.3 小结

本节内容说明了使用 C++ 来写 Node.js 原生扩展的两大理由——性能和开发成本。

对于有显著性能提升的情况，使用 C++ 来完成这一项任务，我们何乐而不为呢？而对于已存在的 C++ 类库，那些难以迁移或者无法迁移的项目我们何苦迁移呢？还有后一种情况的一个小分支，那就是其实并没有已存在的 C++ 类库可以直接用，但是使用 JavaScript 难以实现某种功能。

这两大理由造就了一批又一批的 C++ 原生扩展。

2.1.4 参考资料

- [1] The Computer Language Benchmarks Game: <http://benchmarksgame.alioth.debian.org/>.
- [2] Benchmark of Regex Libraries: <http://lh3lh3.users.sourceforge.net/reb.shtml>.
- [3] Google 发布 CityHash 系列散列算法: <http://www.cnbeta.com/articles/tech/139994.htm>.
- [4] cityhash/src/city.h: <https://github.com/google/cityhash/blob/master/src/city.h>.

[5] 一个由于环境变量产生的 bug: <http://f10.moe/2016/03/07/%E4%B8%80%E4%B8%AA%E7%94%B1%E4%BA%8E%E7%8E%AF%E5%A2%83%E5%8F%98%E9%87%8F%E4%BA%A7%E7%94%9F%E7%9A%84bug/>.

2.2 什么是 C++ 扩展

由于 Node.js 本身就是基于 Chrome V8 引擎和 libuv, 用 C++ 进行开发的, 因此自然能轻而易举对特定结构下使用了特定 API 进行开发的 C++ 代码进行扩展, 使得其在 Node.js 中被 “require” 之后能像调用 JavaScript 函数一样调用 C++ 扩展里面的内容。

本节会从官方文档和 Node.js 的一些源码出发进行解析, 逐步说明 C++ 扩展在 Node.js 中的工作原理, 让读者对其有一个整体的认知, 从而对后续内容的阅读有帮助。

实际上, 很大程度上我们可以将 C++ 扩展和 C++ 模块等同起来。

2.2.1 C++ 模块本质

众所周知, Node.js 是基于 C++ 开发的, 所有底层头文件暴露的 API 也都是适用于 C++ 的。

笔者在 1.1 节中曾提到过, 当我们在 Node.js 中通过 `require()` 载入一个模块的时候, Node.js 运行时会依次枚举后缀名进行寻径, 其中就曾提到后缀名为 `*.node` 的模块, 这是一个 C++ 模块的二进制文件。

实际上, 一个编译好的 C++ 模块除了后缀名是 `*.node` 之外, 它其实就是一个系统的动态链接库。说得直白一点, 这相当于 Windows 下的 `*.dll`、Linux 下的 `*.so` 以及 macOS 下的 `*.dylib`。

我们可以用一些十六进制的编辑器打开一个 `*.node` 的 C++ 模块, 看看其文件头的标识位。

如图 2-2 所示, 这是笔者之前写的阿里云消息队列 ONS 的 Node.js SDK, 图中是该 SDK 的 C++ 模块部分在 Linux 下编译出来的二进制文件的十六进制内容。我们发现它的标识位十六进制是 `0x7F454C46`, 其 ASCII 码所代表的内容是一个不可见空字符 (ASCII 码为 `0x7F`) 后面跟着 “ELF”¹, 这就是一个 Linux 下动态链接库的标识。至于其他系统下编译好的 C++ 模块, 有兴趣的读者可自行验证。

¹ 根据 Man7.org 的相关章节所述, Linux 下的动态链接库文件标识位是 `0x7F454C46`。详细信息可参考 <http://man7.org/linux/man-pages/man5/elf.5.html>。

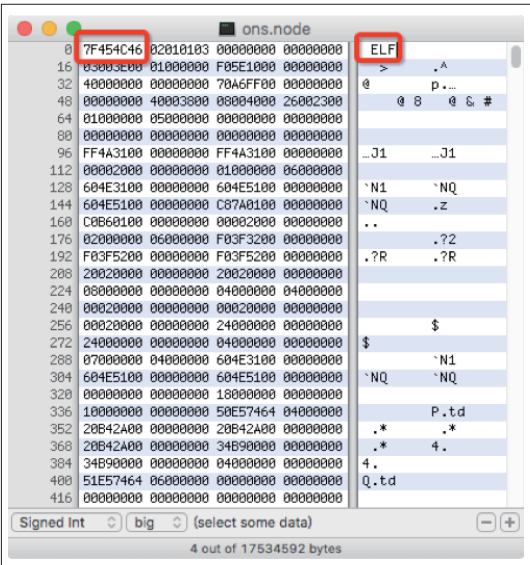


图 2-2 ONS 包中 Linux 下编译的 C++ 模块的十六进制标识位

结合上述说法，我们大概能猜到，在 Node.js 中引入一个 C++ 模块的过程，实际上就是 Node.js 在运行时引入了一个动态链接库的过程。运行时接受 JavaScript 代码中的调用，解析出来具体是扩展中的哪个函数需要被调用，在调用完获得结果之后再通过运行时返回给 JavaScript 代码，如图 2-3 所示。

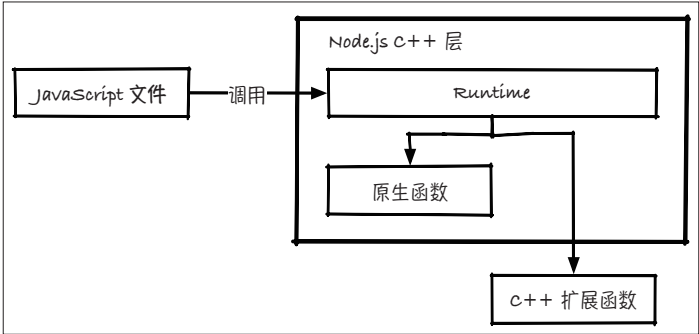


图 2-3 原生函数和 C++ 扩展函数

调用 Node.js 的原生 C++ 函数和调用 C++ 扩展函数的区别就在于前者的代码会直接编译进 Node.js 可执行文件中，而后者的代码则位于一个动态链接库中。

官方文档上有对于 C++ 扩展的说明，我们也可以在后续章节中验证上面的这一猜想。

Node.js C++ 扩展是以 C 或者 C++ 写的动态链接库，可以被 Node.js 以 `require()` 的形式载入，在使用的时候就好像它们就是 Node.js 模块一样。它们主要被用于在 Node.js 的 JavaScript 和 C 或者 C++ 库之间建立起桥梁的关系。

2.2.2 Node.js 模块加载原理

在读者以往写 Node.js 的经验中，Node.js 载入一个源码文件或者一个 C++ 扩展文件是通过 Node.js 中的 `require()` 函数实现的（这里不对 ES6 中的 `import` 做解析）。在第 1 章中笔者也介绍了，这些被载入的文件单位或者粒度就是模块（`module`）了。当然，C++ 模块也被称为 C++ 扩展。

该函数既能载入 Node.js 的内部模块，又能载入开发者的 JavaScript 源码模块以及 C++ 扩展。本节就对这 3 种类型模块的载入原理进行解读，让读者有一个更进一步的了解。

在阅读本节的时候，读者可以只看本书，也可以跟随本书的脚步打开 Node.js 的 Git 仓库 6.9.4 版本的源码，一起进行解读。

代码地址如下：<https://github.com/nodejs/node/tree/v6.9.4>。

1. Node.js 入口

首先笔者先解析 Node.js 的入口。在 Node.js 6.9.4 版本中，Node.js 在 C++ 代码层面的入口在其源码的 `src/node_main.cc` 文件中。

```
// UNIX
int main(int argc, char *argv[]) {
    // Disable stdio buffering, it interacts poorly with printf()
    // calls elsewhere in the program (e.g., any logging from V8.)
    setvbuf(stdout, nullptr, _IONBF, 0);
    setvbuf(stderr, nullptr, _IONBF, 0);
    return node::Start(argc, argv);
}
```

上述代码说明了进入 C++ 主函数之后直接调用 `node` 这个命名空间中的 `Start` 函数，而这个函数则处于 `src/node.cc`。

根据 `src/node.cc` 中 `Start` 函数的逐层深入，我们在 `LoadEnvironment` 函数中会发现如下这段代码。

```
Local<String> script_name = FIXED_ONE_BYTE_STRING(env->isolate(),
                                                    "bootstrap_node.js");
Local<Value> f_value = ExecuteString(env, MainSource(env), script_name);
```



```

...

Local<Function> f = Local<Function>::Cast(f_value);

...

f->Call(Null(env->isolate()), 1, &arg);

```

这段代码的意思是 Node.js 执行 `lib/internal/bootstrap_node.js` 文件以进行初始化启动，这里还没有 `require` 的概念。文件中的源码没有经过 `require()` 函数进行闭包化操作，所以执行该文件之后得到的 `f_value` 就是这个 `bootstrap_node.js` 文件中所实现的那个函数对象。

由于 V8 的值（包括对象、函数等）均继承自 `Value` 基类，这里在得到函数的 `Value` 实例之后需要将其转化成能用的 `Function` 对象¹，然后以 `env->process_object()` 为参数执行这个从 `bootstrap_node.js` 中得到的函数。

分析了上面的代码，我们大概了解了 Node.js 入口启动的流程，如图 2-4 所示。

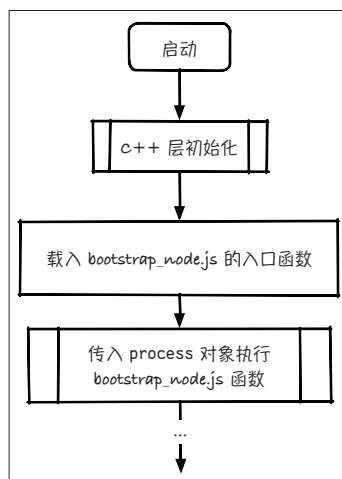


图 2-4 Node.js 入口启动的流程

2. process 对象

前面笔者提到了执行 Node.js 初始化函数时会传入 `env->process_object()`，而对应 `lib/internal/bootstrap_node.js` 文件中这个参数的含义其实就是 `process` 对象。

¹ 有关 V8 的相关内容会在第 3 章中详细介绍。

这个 `process` 对象就是 Node.js 中大家经常用到的全局对象 `process`。具体的一些公共 API 可以在 Node.js 官方文档的 `process` 一节¹中查阅。

好的，现在抛开 Node.js 文档，回到 `process` 中来。这个 `env->process_object()` 的一些内容就是在 `src/node.cc` 中实现的。我们能很容易追踪到这个文件中的 `SetupProcessObject` 函数。

篇幅所限，这里就不列出这个函数的所有代码了。下面列出其部分设置让读者感受一下温暖，这是来自 Node.js 的关怀。

```
env->SetMethod(process, "hrtime", Hrtime);
env->SetMethod(process, "cpuUsage", CPUUsage);
env->SetMethod(process, "dlopen", DLOpen);
env->SetMethod(process, "uptime", Uptime);
env->SetMethod(process, "memoryUsage", MemoryUsage);
env->SetMethod(process, "binding", Binding);
```

读者是不是觉得上面列举的方法、属性设置更熟悉了？除了 `binding` 和 `dlopen` 之外，其他几个都是 Node.js 文档中原原本本列出的 `process` 对象中暴露的 API 内容。关于这些函数的具体实现，有兴趣的读者可以自行翻阅 Node.js 的源码，毕竟本书不叫《Node.js 源码解析》。

3. 几种模块的加载过程

前面介绍了一些 Node.js 入口相关的内容之后，笔者接下来要将模块分 4 种类型，介绍其加载的过程。

这 4 种模块如下：

- C++ 核心模块；
- Node.js 内置模块；
- 用户源码模块；
- C++ 扩展。

C++ 核心模块和 Node.js 内置模块属于 1.1.2 节中提到过的 Node.js 核心模块；而用户源码模块和 C++ 扩展模块属于文件模块。

(1) C++ 核心模块

C++ 核心模块在 Node.js 源码中其实就是采用纯 C++ 编写的，并未经过任何 JavaScript 代码封装过的原生模块，其有点类似于本书所介绍的 C++ 扩展，而区别在于前者存在于 Node.js 源码中并且编译进 Node.js 的可执行二进制文件中，后者则以动态链接库的形式存在。

¹ 文档地址：<https://nodejs.org/docs/v6.9.4/api/process.html>。

在介绍 C++ 核心模块的加载过程之前，笔者先提一下前面出现过的 `process.binding` 函数。它对应的是 `src/node.cc` 文件中的 `Binding` 函数。姑且不管这个函数在哪里被用到，笔者先对源码进行一遍粗略的解析。

```
static void Binding(const FunctionCallbackInfo<Value>& args) {
    Environment* env = Environment::GetCurrent(args);

    Local<String> module = args[0]->ToString(env->isolate());
    node::Utf8Value module_v(env->isolate(), module);

    Local<Object> cache = env->binding_cache_object();
    Local<Object> exports;

    if (cache->Has(env->context(), module).FromJust()) {
        exports = cache->Get(module)->ToObject(env->isolate());
        args.GetReturnValue().Set(exports);
        return;
    }

    // 将一个字符串附加在 process.moduleLoadList 后面
    char buf[1024];
    snprintf(buf, sizeof(buf), "Binding %s", *module_v);

    Local<Array> modules = env->module_load_list_array();
    uint32_t l = modules->Length();
    modules->Set(l, OneByteString(env->isolate(), buf));

    node_module* mod = get_builitn_module(*module_v);
    if (mod != nullptr) {
        exports = Object::New(env->isolate());
        // 内置的模块对象并没有 "module" 对象，只有 exports
        CHECK_EQ(mod->nm_register_func, nullptr);
        CHECK_NE(mod->nm_context_register_func, nullptr);
        Local<Value> unused = Undefined(env->isolate());
        mod->nm_context_register_func(exports, unused,
            env->context(), mod->nm_priv);
        cache->Set(module, exports);
    } else if (!strcmp(*module_v, "constants")) {
        exports = Object::New(env->isolate());
        DefineConstants(env->isolate(), exports);
        cache->Set(module, exports);
    } else if (!strcmp(*module_v, "natives")) {
        exports = Object::New(env->isolate());
        DefineJavaScript(env, exports);
        cache->Set(module, exports);
    }
}
```

```

    } else {
        char errmsg[1024];
        snprintf(errmsg,
                 sizeof(errmsg),
                 "No such module: %s",
                 *module_v);
        return env->ThrowError(errmsg);
    }

    args.GetReturnValue().Set(exports);
}

```

其中 `Local<String> module = args[0]->ToString(env->isolate());` 和 `node::Utf8Value module_v(env->isolate(), module);` 两句代码意味着从参数中获得文件标识（或者也可以认为是文件名）的字符串并赋值给 `module_v`。

在得到标识字符串之后，Node.js 将通过 `node_module* mod = get_builtin_module(*module_v);`¹ 这句代码获取 C++ 核心模块，例如未经源码 lib 目录下的 JavaScript 文件封装的 `file` 模块²。我们注意到这里获取核心模块用的是一个 `get_builtin_module` 函数，这个函数内部做的工作就是在一个名为 `modlist_builtin` 的 C++ 核心模块链表上对比文件标识，从而返回相应的模块。

追根溯源，这些 C++ 核心模块则是在 `node_module_register` 函数中被逐一注册进链表中的，我们可以阅读一下下面的代码。

```

static node_module* modpending;
static node_module* modlist_builtin;
static node_module* modlist_linked;
static node_module* modlist_addon;

...

extern "C" void node_module_register(void* m) {
    struct node_module* mp = reinterpret_cast<struct node_module*>(m);

    if (mp->nm_flags & NM_F_BUILTIN) {
        mp->nm_link = modlist_builtin;
        modlist_builtin = mp;
    }
}

```

1 在 Node.js 中，`Utf8Value` 类的星号操作符重载的返回值是 `char*` 指针类型（即字符串），可以参考源码文件：<https://github.com/nodejs/node/blob/v6.9.4/src/util.h#L264-L363>。

2 `file` 的 C++ 核心模块源码文件是 `src/node_file.cc`，可以阅读该文件源码：https://github.com/nodejs/node/blob/v6.9.4/src/node_file.cc。

```

    } else if (!node_is_initialized) {
        // "Linked" modules are included as part of the node project.
        // Like builtins they are registered *before* node::Init runs.
        mp->nm_flags = NM_F_LINKED;
        mp->nm_link = modlist_linked;
        modlist_linked = mp;
    } else {
        modpending = mp;
    }
}

struct node_module* get_builtin_module(const char* name) {
    struct node_module* mp;

    for (mp = modlist_builtin; mp != nullptr; mp = mp->nm_link) {
        if (strcmp(mp->nm_modname, name) == 0)
            break;
    }

    CHECK(mp == nullptr || (mp->nm_flags & NM_F_BUILTIN) != 0);
    return (mp);
}

```

这个 `node_module_register` 函数清晰表达了，如果传入待注册模块的标识位是内置模块 (`mp->nm_flags & NM_F_BUILTIN`)，就将其加入 C++ 核心模块的链表中；否则将认为它是其他模块，由于这个条件分支与本书关联性不大，笔者对后者就不进行深究了。

我们继续对 C++ 核心模块分析下去。在 `src/node.h` 中有一个宏是用于注册 C++ 核心模块的。

```

#define NODE_MODULE_CONTEXT_AWARE_X(modname, regfunc, priv, flags) \
extern "C" { \
    static node::node_module _module = \
    { \
        NODE_MODULE_VERSION, \
        flags, \
        NULL, \
        __FILE__, \
        NULL, \
        (node::addon_context_register_func) (regfunc), \
        NODE_STRINGIFY(modname), \
        priv, \
        NULL \
    }; \
    NODE_C_CTOR(_register_ ## modname) { \
        node_module_register(&_module); \
    } \
}

```

```

    }
}

#define NODE_MODULE(modname, regfunc) \
    NODE_MODULE_X(modname, regfunc, NULL, 0)

#define NODE_MODULE_CONTEXT_AWARE(modname, regfunc) \
    NODE_MODULE_CONTEXT_AWARE_X(modname, regfunc, NULL, 0)

#define NODE_MODULE_CONTEXT_AWARE_BUILTIN(modname, regfunc) \
    NODE_MODULE_CONTEXT_AWARE_X(modname, regfunc, NULL, NM_F_BUILTIN) \

```

结合之前看的 `node_module_register` 函数和这个 `src/node.h` 中的宏定义，我们发现只要 Node.js 在其 C++ 源码中调用 `NODE_MODULE_CONTEXT_AWARE_BUILTIN` 这个宏，就有一个模块会被注册进 Node.js 的 C++ 核心模块链表中。

那么问题来了：什么时候会有这样的注册呢？读者不妨自己动手，到之前提到过的 `file` 模块看看吧。它的源码是 `src/node_file.cc`，这里的最后一行就是答案了。

```
NODE_MODULE_CONTEXT_AWARE_BUILTIN(fs, node::InitFs)
```

这个宏被展开后的结果将会是这样的：

```

extern "C" {
    static node::node_module _module =
    {
        NODE_MODULE_VERSION,
        NM_F_BUILTIN,
        NULL,
        __FILE__,
        NULL,
        (node::addon_context_register_func) (node::InitFs),
        NODE_STRINGIFY(fs),
        NULL,
        NULL
    };

    NODE_C_CTOR(_register_ ## fs) {
        node_module_register(&_module);
    }
}

```

至此，真相大白。也就是说，基本上在每个 C++ 核心模块的源码末尾都有一个宏调用将该模块注册进 C++ 核心模块的链表中，以供执行 `process.binding` 时进行获取。

(2) Node.js 内置模块

Node.js 内置模块基本上等同于其官方文档中放出来的那些模块。这些模块大多是在源码 lib 目录下以同名 JavaScript 代码的形式被实现，而且很多 Node.js 内置模块实际上都是对 C++ 核心模块的一个封装。

如 lib/crypto.js 中就有一段 `const binding = process.binding("crypto");` 这样的代码，它的很多内容都是基于 C++ 核心模块中的 crypto 进行实现的。

说到这里，大家可能有一个疑问，为什么明明在 Node.js 源码下面有一个 lib 目录，并且里面有一堆堆的 JavaScript 代码，如 net、fs 等，为什么在通过下载、安装或者编译好后就只有一个单独的二进制可执行文件了呢，难道 JavaScript 代码也能被编译到 Node.js 的可执行文件吗？

说得一点儿也没错，这些 lib 下的 JavaScript 文件的确被编译进 Node.js 的可执行文件了。下面笔者会一一道来。

请把注意力转移至 Node.js 的启动脚本 lib/internal/bootstrap_node.js 中。其代码的最下面位置有一个 NativeModule 类的声明。为了更突出关键代码，本书中的该部分源码略去了特殊情况分支和缓存的处理。

```
function NativeModule(id) {
  this.filename = `${id}.js`;
  this.id = id;
  this.exports = {};
  this.loaded = false;
  this.loading = false;
}

NativeModule._source = process.binding('natives');
NativeModule._cache = {};

NativeModule.require = function(id) {
  if (id === 'native_module') {
    return NativeModule;
  }

  ...

  process.moduleLoadList.push(`NativeModule ${id}`);

  const nativeModule = new NativeModule(id);

  ...
```

```

    nativeModule.compile();

    return nativeModule.exports;
};

...

NativeModule.getSource = function(id) {
    return NativeModule._source[id];
};

NativeModule.wrap = function(script) {
    return NativeModule.wrapper[0] + script + NativeModule.wrapper[1];
};

NativeModule.wrapper = [
    '(function (exports, require, module, __filename, __dirname) { ',
    '\n});'
];

NativeModule.prototype.compile = function() {
    var source = NativeModule.getSource(this.id);
    source = NativeModule.wrap(source);

    this.loading = true;

    try {
        const fn = runInThisContext(source, {
            filename: this.filename,
            lineOffset: 0,
            displayErrors: true
        });
        fn(this.exports, NativeModule.require, this, this.filename);

        this.loaded = true;
    } finally {
        this.loading = false;
    }
};

```

这个 `NativeModule` 就是 Node.js 内置模块的相关处理类了。它有一个叫 `require` 的静态函数，当其参数 `id` 值为 `'native_module'` 时返回的是它本身，否则就进入 `nativeModule.compile` 进行编译。

进而把目光转向 `compile` 函数，它的第一行代码就是获取该模块的源码。

注意，接下来就是我们本节最开始提出的疑问的答案剖析。

源码是通过 `NativeModule.getSource` 获取的，`NativeModule.getSource` 函数返回的是 `NativeModule._source` 数组中的相应内容。

那么，这个 `NativeModule._source` 是哪里来的呢？

`NativeModule._source = process.binding('natives');` 这一行代码说明了 `NativeModule._source` 的出处。

前面笔者详细介绍的 `process.binding` 函数在这里派上用场了。

我们回过头去仔细看一下 `src/node.cc` 中 `Binding` 的源码，其中有一段判断是这样的。

```
...
} else if (!strcmp(*module_v, "natives")) {
    exports = Object::New(env->isolate());
    DefineJavaScript(env, exports);
    cache->Set(module, exports);
} else {
...

```

也就是说执行 `process.binding('natives')` 返回的结果是 `DefineJavaScript` 函数中处理的内容。

马不停蹄来到了 `src/node_javascript.cc` 中，让我们好好观察一下这个 `DefineJavaScript`。

```
void DefineJavaScript(Environment* env, Local<Object> target) {
    HandleScope scope(env->isolate());

    for (auto native : natives) {
        if (native.source != internal_bootstrap_node_native) {
            Local<String> name = String::NewFromUtf8(env->isolate(), native.name);
            Local<String> source =
                String::NewFromUtf8(
                    env->isolate(), reinterpret_cast<const char*>(native.source),
                    NewStringType::kNormal, native.source_len).ToLocalChecked();
            target->Set(name, source);
        }
    }
}

```

从上述代码中我们了解到了它做的事情就是遍历一遍 `natives` 数组里面的内容，并将其一一加入要返回的对象中，其中对象名的键名为源码文件名标识，键值是源码本体的字符串。

现在能走到这一步已经很不容易了。细心的读者会发现逛遍整个项目都找不到这个 `natives` 在哪里。

让我们放开“脑洞”想一想，所有的 Node.js 内置模块本来“一个萝卜一个坑”地在 `lib` 目录下好好待着，但是到这边载入的时候却在 Node.js 的 C++ 源码中以 `natives` 变量的形式存在——这中间发生了什么？

其实说来也简单——这一层是在编译时做的。

请打开 Node.js 的 GYP 配置文件 `node.gyp`。

其中有一步（也就是有一个目标配置）是 `node_js2c`，在这一步中做的事情就是用 Python 去调用一个名为 `tools/js2c.py` 的文件。而这个 `js2c.py` 就是问题的关键所在了。

这是一个 Python 脚本，主要的作用是将 `lib` 下的 JavaScript 文件转换成 `src/node_natives.h` 文件。

熟悉 Python 的读者可以自行挖掘一下该文件的具体实现，由于篇幅的原因本书就不展开详述了。

这个 `src/node_natives.h` 文件会在 Node.js 编译前完成，这样在编译到 `src/node_javascript.cc` 时它所需要的 `src/node_natives.h` 头文件就存在了。

`src/node_natives.h` 源文件经过 `js2c.py` 转换后，会以类似于下述代码的形式存在。

```
namespace node {
  const char node_native[] = {47, 47, 32, 67, 112 ...}

  const char console_native[] = {47, 47, 32, 67, 112 ...}

  const char buffer_native[] = {47, 47, 32, 67, 112 ...}

  ...
}

struct _native {const char name;  const char* source;  size_t source_len;};

static const struct _native natives[] = { { "node", node_native,
sizeof(node_native)-1 },

  {"dgram", dgram_native, sizeof(dgram_native)-1 },
```

```

{"console", console_native, sizeof(console_native)-1 },

{"buffer", buffer_native, sizeof(buffer_native)-1 },

...
}

```

在此可以看出，这样的文件形式正好满足了 `src/node_javascript.cc` 中 `DefineJavaScript` 函数所需要的 `natives` 格式。

也就是说，在 `Node.js` 中调用 `NativeModule.require` 的时候，会根据传入的文件标识来返回相应的 JavaScript 源文件内容，如 `"dgram"` 对应的是 `lib/dgram.js` 中的 JavaScript 代码字符串。

把传说中编译进 `Node.js` 二进制文件的 JavaScript 代码的神秘面纱揭开以后，我们现在回到 `NativeModule.compile` 函数中来。它会在刚获取到的内置模块 JavaScript 源码字符串前后用 `(function (exports, require, module, __filename, __dirname) { 和 });` 进行包裹，形成一段闭包代码。之后将其放入 `vm`¹ 中运行，并传入事先准备好的 `module` 和 `exports` 对象供其导出。

如此一来，内置模块就完成了加载。

(3) 用户源码模块

用户源码模块指的是用户在项目中的 `Node.js` 源码，以及所使用的第三方包中的模块²。一句话概括，就是非 `Node.js` 内置模块的 JavaScript 源码模块。

这些模块是在程序运行时，在需要被使用的时候按需被 `require()` 函数加载的。

与内置模块类似，每个用户源码模块会被加上一个闭包的头尾，然后 `Node.js` 执行这个闭包产生结果。

我们打开 `lib/module.js` 这个内置模块可以找到其细节上的实现。我们平时在源码中执行的 `require()` 函数其实就是这个 `Module` 类实例对象的 `require()` 函数。

一个 `Module` 类的实例对象就是一个用户源码模块本体，用户通过 `require()` 所引入的文件代码及其在 `vm` 沙盒中的结果就是这个模块的核心。只不过我们日常稍微模糊化了这个 `Module` 和用户源码的概念，把它们都称作模块。

¹ `vm` 模块主要用于创建独立运行的沙箱体制，通过 `vm`，JavaScript 源码可以被编译后立即执行或者编译保存下来稍后执行。它是 `Node.js` 里面的核心模块，支撑了 `require` 方法和 `Node.js` 的运行机制，我们有些时候可能也要用到 `vm` 模板来做一些特殊的事情。

² 包和模块的定义在第 1 章曾提及。

我们在平时写 Node.js 代码时经常用到的 `module.exports` 的 `module`，指的就是 `Module` 类的实例对象，`exports` 就是这个对象中的一个部分。当我们写 `module.exports = foo` 的时候就是对这个 `module` 对象的 `exports` 变量重新赋值。

```
Module.prototype.require = function(path) {
  assert(path, 'missing path');
  assert(typeof path === 'string', 'path must be a string');
  return Module._load(path, this, /* isMain */ false);
};
```

`require()` 直接调用了 `Module._load` 这个静态函数，并声明 `isMain`（是否是入口模块）为 `false`。

```
Module._load = function(request, parent, isMain) {
  ...

  var filename = Module._resolveFilename(request, parent, isMain);

  var cachedModule = Module._cache[filename];
  if (cachedModule) {
    return cachedModule.exports;
  }

  if (NativeModule.nonInternalExists(filename)) {
    return NativeModule.require(filename);
  }

  var module = new Module(filename, parent);

  if (isMain) {
    process.mainModule = module;
    module.id = '.';
  }

  Module._cache[filename] = module;

  tryModuleLoad(module, filename);

  return module.exports;
};
```

`Module._load` 中大致分了几步走，具体流程如图 2-5 所示。

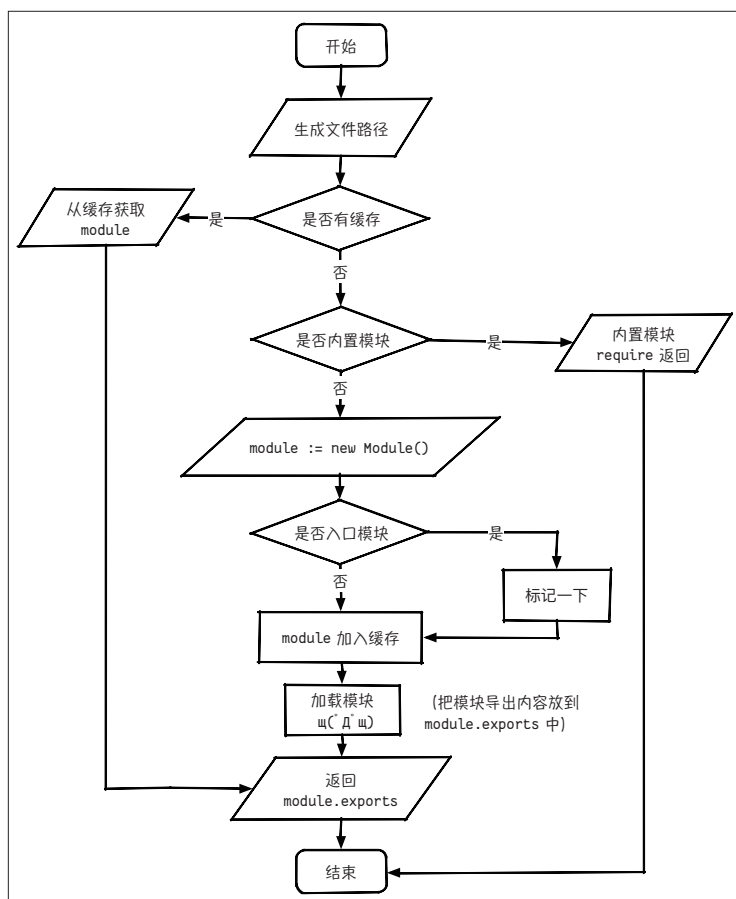


图 2-5 Module._load 流程图

在流程图中笔者简化了“加载模块”这个步骤，因为在 `_load` 函数中，它是作为另一个函数被调用的——`tryModuleLoad(module, filename)`。顾名思义，`tryModuleLoad` 是尝试载入模块的意思，其实它就是在执行 `module.load` 时多加了一些错误处理的过程，本体其实还是 `module` 对象的 `load` 函数。

```

Module.prototype.load = function(filename) {
  this.filename = filename;
  this.paths = Module._nodeModulePaths(path.dirname(filename));

  var extension = path.extname(filename) || '.js';
  if (!Module._extensions[extension]) extension = '.js';
  Module._extensions[extension](this, filename);
  this.loaded = true;
};

```

`load()` 函数的源码相当于一个适配器，其根据传进来文件名的后缀名不同，会使用不同的载入规则。默认情况下，有 3 种规则：

- `Module._extensions[".js"]`
- `Module._extensions[".json"]`
- `Module._extensions[".node"]`

本节将介绍 `Module._extensions[".js"]` 这种规则。该规则做的事情分两步：

- ① 同步读取源码（`filename`）的内容，使用 `fs.readFileSync`；
- ② 调用 `module._compile()` 函数编译源码并执行。

这其中的第二步就很有讲究。下面先看看 `module._compile()` 代码。

```
Module.prototype._compile = function(content, filename) {
  ...

  var wrapper = Module.wrap(content); // 生成闭包源码

  var compiledWrapper = vm.runInThisContext(wrapper, {
    filename: filename,
    lineOffset: 0,
    displayErrors: true
  });

  ...

  var dirname = path.dirname(filename);
  var require = internalModule.makeRequireFunction.call(this);
  var args = [this.exports, require, this, filename, dirname];
  var depth = internalModule.requireDepth;
  if (depth === 0) stat.cache = new Map();
  var result = compiledWrapper.apply(this.exports, args);
  if (depth === 0) stat.cache = null;
  return result;
};
```

其实这个函数与前面提到的 `NativeModule` 的 `_compile` 函数类似，都是生成闭包源码，然后传入相应的函数执行，流程如图 2-6 所示。

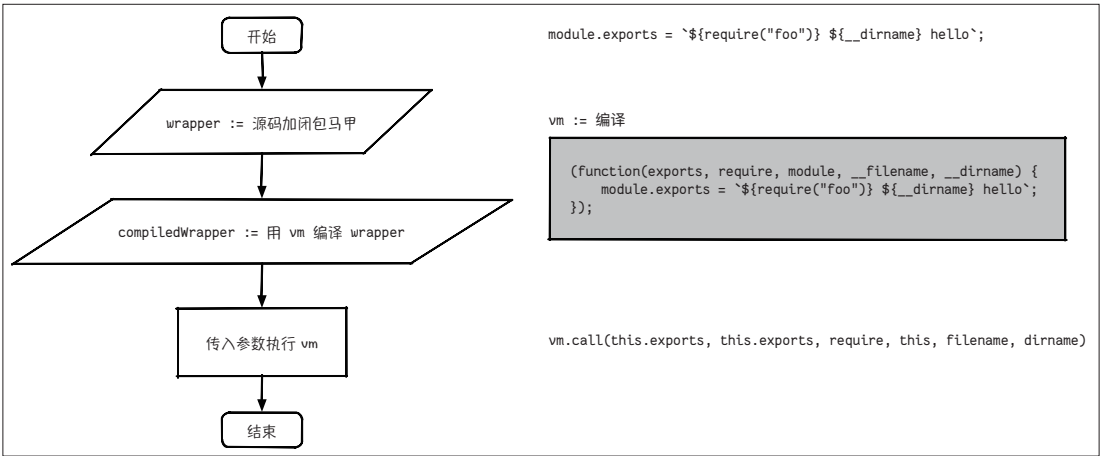


图 2-6 Module.prototype._compile 流程图

一个模块的源码经过闭包化之后，就形成了一个接收 exports、require、module、__filename 和 __dirname 的闭包函数。这就是我们平时编写代码的时候能直接使用 exports、module、require 等内容的原因了。在我们编写的源码模块被载入的时候，这些变量会随着闭包传进来而被使用。这个闭包会在第一次加载该模块的时候执行一次，之后就一直存在于模块缓存中（除非手动清除缓存），这就解开了我们在第 1 章中提到过的一个模块逻辑代码只会被执行一次的疑惑。

在 1.1.2 节中写道：实际上在 Node.js 运行中，通常情况下一个包一旦被加载了，那么在第二次执行 require() 的时候就会在缓存中获取暴露的 API，而不会重新加载一遍该模块里面的代码再次返回。

值得注意的是，传进来的 module 就是笔者本节所讲的 Module 类的对象实例，所以我们对 module.exports 赋值实际上就是对这个传入的 module 对象进行赋值；传进来的 require 就是经包装后的 Module.prototype.require，其在某种意义上等同于 Module._load。

现在笔者再梳理一下用户源码模块的载入流程。

- ① 开发者调用 require()（这在某种意义上等同于调用 Module._load）。
- ② 闭包化对应文件的源码，并传入相关参数执行（若有缓存，则直接返回）。
- ③ 通常在执行过程中 module.exports 或者 exports 会被赋值。
- ④ Module.prototype._load 在最后返回这个模块的 exports 给上游。

入口模块

在我们以非 REPL 形式执行 Node.js 的时候，通常会使用 `node <文件名>` 的命令来启动一个 Node.js 程序。而这个指定的文件就是一个入口模块了。入口模块其实也是用户源码模块的一种，只不过它将作为程序入口被执行而已。

在 `src/node_main.cc` 中，启动 Node.js 的一行代码是 `node::Start(argc, argv)`，上面提到的文件名就会在 `argv` 这个数组中被传进 `Start` 函数中。辗转之后，被处理后的 `argv` 会被传到 `NodeInstanceData` 类的构造函数中。

```
// NodeInstanceData 类的声明
class NodeInstanceData {
public:
    NodeInstanceData(NodeInstanceType node_instance_type,
                    ...

private:
    const NodeInstanceType node_instance_type_;
    int exit_code_;
    uv_loop_t* const event_loop_;
    const int argc_;
    const char** argv_;
    const int exec_argc_;
    const char** exec_argv_;
    const bool use_debug_agent_flag_;

    ...
};
```

在接下来的 `StartNodeInstance()` 函数中以及后面的几度转手中，里面的 `argv` 和 `exec_argv` 会被传到 `process` 对象中供 `lib/internal/bootstrap_node.js` 使用。

```
void SetupProcessObject(Environment* env,
                        int argc,
                        const char* const* argv,
                        int exec_argc,
                        const char* const* exec_argv) {

    ...

    // process.argv
    Local<Array> arguments = Array::New(env->isolate(), argc);
    for (int i = 0; i < argc; ++i) {
        arguments->Set(i, String::NewFromUtf8(env->isolate(), argv[i]));
    }
    process->Set(FIXED_ONE_BYTE_STRING(env->isolate(), "argv"),
                arguments);
```



```
...
}
```

至于 `argv` 和 `exec_argv` 的区别，读者可以看看 `src/node_main.cc` 里面 `ParseArgs` 函数中的具体实现，这里不再赘述。

我们在 `lib/internal/bootstrap_node.js` 中可以看出，整个闭包函数被执行的时候，会执行里面的 `startup()` 函数，这样才算进入了 Node.js 的 JavaScript 代码启动的流程。

该函数中提及了，如果是正常以 `node <文件名>` 的形式启动，会进入这么一段逻辑：

```
...

} else if (process.argv[1]) {
  // 使 process.argv[1] 变成一个完整路径
  const path = NativeModule.require('path');
  process.argv[1] = path.resolve(process.argv[1]);

  const Module = NativeModule.require('module');

  ...

  run(Module.runMain);
} else {
  ...
```

也就是说取出文件名（即 `process.argv[1]`）并将其格式化成为绝对路径，然后执行 `run()` 进行启动。通常情况下这个 `run()` 函数会立即执行以参数形式传进去的函数，也就是执行 `Module.runMain()`。

`Module` 就是 `lib/module.js` 这个内置模块。

```
Module.runMain = function() {
  Module._load(process.argv[1], null, true);
  process._tickCallback();
};
```

我们可以看到这里调用 `Module._load` 时传的参数与通过 `require()` 传入的参数相比略有不同。第二个参数 `parent` 变为 `null`，因为入口文件上面没有父模块了；第三个参数表示入口文件的布尔型参数也变成 `true`，这个参数一旦为 `true`，在 `module` 对象生成之后，它会被顺便赋值到 `process.mainModule`。

这个函数直接加载执行了命令行中指定的文件，并且执行和清空第一次 `nextTick` 中的一些回调。以上就是入口模块的加载过程。

(4) C++ 扩展

根据前面所述,用户源码模块与 C++ 扩展模块加载时的区别仅仅是在 `Module.prototype.load` 函数中进行区分的。我们可以再回过头来看规则,一个是 `Module._extensions[".js"]`,而另一个是 `Module._extensions[".node"]` (这里我们忽略 *.json 文件)。

也就是说,我们如果将一个 C++ 扩展模块作为 Node.js 入口文件,理论上也是可以的。毕竟 Node.js 入口模块的执行函数 `Module.runMain` 是通过调用函数 `Module._load(process.argv[1], null, true)` 来完成的。

想用 C++ 扩展模块作为入口文件进行尝试的读者也可以进入随书源码的“2. cpp entry”目录进行取证。

进入“2. cpp entry”目录,并依次执行下面的命令:

```
$ node-gyp configure
$ node-gyp build
```

在万事俱备之后,执行 `$ node build/Release/entry.node` 会有下面的结果:

```
$ node build/Release/entry.node
Module {
  id: '.',
  exports: { runCallback: [Function: runCallback] },
  parent: null,
  filename: '/Users/USER/2. cpp entry/build/Release/entry.node',
  loaded: false,
  children: [],
  paths:
    [ '/Users/USER/2. cpp entry/build/Release/node_modules',
      '/Users/USER/2. cpp entry/build/node_modules',
      '/Users/USER/2. cpp entry/node_modules',
      '/Users/USER/node_modules',
      '/Users/node_modules',
      '/node_modules' ] } '---' { runCallback: [Function: runCallback] }
```

由上面的执行结果我们可以看出, `node <C++ 扩展模块>` 的命令也可以正常执行,效果跟用户源码模块并无两样。该目录下 C++ 源码的意思转义为 Node.js 源码大致如下:

```
function runCallback(...) {
  ...
}

exports.runCallback = runCallback;

console.log(module, "---", exports);
```

所以输出如上命令行的结果也是在意料之中的。

笔者接下来剖析一下这个加载 *.node 扩展的函数——请打开 lib/module.js。

```
Module._extensions['.node'] = function(module, filename) {
    return process.dlopen(module, path._makeLong(filename));
};
```

简而言之，载入 *.node 的 C++ 扩展直接使用了 process.dlopen 函数。dlopen 是在 src/node.cc 中的 SetupProcessObject 里面被挂载上去的，它实际上对应的函数是这个 src/node.cc 文件中的 DLOpen 函数。

```
void DLOpen(const FunctionCallbackInfo<Value>& args) {
    Environment* env = Environment::GetCurrent(args);
    uv_lib_t lib;

    ...

    Local<Object> module = args[0]->ToObject(env->isolate());
    node::Utf8Value filename(env->isolate(), args[1]);

    // 使用 uv_dlopen 加载链接库
    const bool is_dlopen_error = uv_dlopen(*filename, &lib);
    node_module* const mp = modpending;
    modpending = nullptr;

    ...

    // 将加载的链接库句柄转移到 mp 上
    mp->nm_dso_handle = lib.handle;
    mp->nm_link = modlist_addon;
    modlist_addon = mp;

    Local<String> exports_string = env->exports_string();

    // exports_string 其实就是 `"exports"`
    // 这句的意思是 `exports = module.exports`
    Local<Object> exports = module->Get(exports_string)->ToObject(env->isolate());

    if (mp->nm_context_register_func != nullptr) {
        mp->nm_context_register_func(exports, module, env->context(),
        mp->nm_priv);
    } else if (mp->nm_register_func != nullptr) {
        mp->nm_register_func(exports, module, mp->nm_priv);
    } else {
        uv_dlclose(&lib);
    }
}
```

```

    env->ThrowError("Module has no declared entry point.");
    return;
}
}

```

`DLOpen` 函数先使用 `uv_dlopen` 函数打开了 `*.node` 扩展（也就是动态链接库），将其载入到内存 `uv_lib_t` 中。

```

// uv_lib_t 数据结构
typedef struct {
    void* handle; // Windows 下为 HMODULE
    char* errmsg;
} uv_lib_t;

```

然后通过 `mp->nm_dso_handle` 将使用 `uv_dlopen` 加载的动态链接库句柄转移到 `node_module` 结构体的实例对象上来。

```

// node_module 数据结构
struct node_module {
    int nm_version;
    unsigned int nm_flags;
    void* nm_dso_handle;
    const char* nm_filename;
    node::addon_register_func nm_register_func;
    node::addon_context_register_func nm_context_register_func;
    const char* nm_modname;
    void* nm_priv;
    struct node_module* nm_link;
};

```

在这个 `DLOpen` 函数中的后续内容我们先放一下，这里看一下 `uv_dlopen` 加载 `*.node` 扩展的时候发生了什么事吧。

首先这个 `entry.node` 所对应的源码 `entry.cpp` 中有如下代码：

```

NODE_MODULE(addon, init)

```

这是一个宏，类似的宏在前面 C++ 核心模块相关内容中介绍过。这就是将一个模块注册进 Node.js 的模块列表。这个宏在展开后的逻辑是去执行 `src/node.cc` 里的 `node_module_register` 函数。为了方便阅读，这里再给出该函数相关的逻辑代码。

```

extern "C" void node_module_register(void* m) {
    struct node_module* mp = reinterpret_cast<struct node_module*>(m);

```

```

if (mp->nm_flags & NM_F_BUILTIN) {
    ...
} else if (!node_is_initialized) {
    ...
} else {
    modpending = mp;
}
}

```

当然，通常情况下 Node.js 是已经初始化好了的，所以不会进入 `!node_is_initialized` 这个条件分支；而且一个 C++ 扩展显然不是一个内置的模块，那么 `mp->nm_flags & NM_F_BUILTIN` 也不成立。最后，就只剩下 `modpending = mp;` 这个逻辑了。也就是说，把由 C++ 扩展（*.node 文件）中注册的模块赋值给 `modpending`，看变量名我们就知道这是一个注册好的待处理的模块。

弄明白了前面说的这一点，我们就知道了在 `uv_dlopen` 函数执行的时候发生了什么事情——加载 *.node 模块（由于 `NODE_MODULE` 宏将模块赋值给 `modpending`）。

那么在 `DLOpen` 的后续逻辑中我们的思路就清晰起来了。在 `uv_dlopen` 之后有这样的两句代码：

```

node_module* const mp = modpending;
modpending = nullptr;

```

就是把刚加载赋值好的 `modpending` 取出来赋值给 `mp`，并将 `modpending` 指向一个空指针（以免发生野指针等情况）。

好了，`mp` 就是刚通过 `uv_dlopen` 加载进来的动态链接库通过 `NODE_MODULE` 宏生成的模块对象处理器了。不过，这个模块对象处理器还只是空壳，需要将 `module` 和 `exports` 两个对象传进去才能把要导出的内容挂载上去，这就跟先前的用户源码模块编译一样。

于是接下去的步骤就是将 `exports` 和 `module` 挂载上导出内容。

在“2. cpp entry”中，我们展开 `NODE_MODULE` 之后可以得知，`nm_register_func` 就是 `init` 函数，所以会进入下面的一个分支条件中执行，也就是把 `module` 和 `exports` 两个对象传给 `init` 函数执行。

```

if (mp->nm_register_func != nullptr) {
    mp->nm_register_func(exports, module, mp->nm_priv);
}

```

而通过阅读“2. cpp entry”中的 init 函数我们可以得知，在这里把 RunCallback 函数挂载到 exports 对象上，并且调用 console.log 输出了 module 和 exports 两个对象。

为了更清晰地展示 DLOpen 函数的流程，笔者画了一个关于 DLOpen 函数的流程图，如图 2-7 所示。

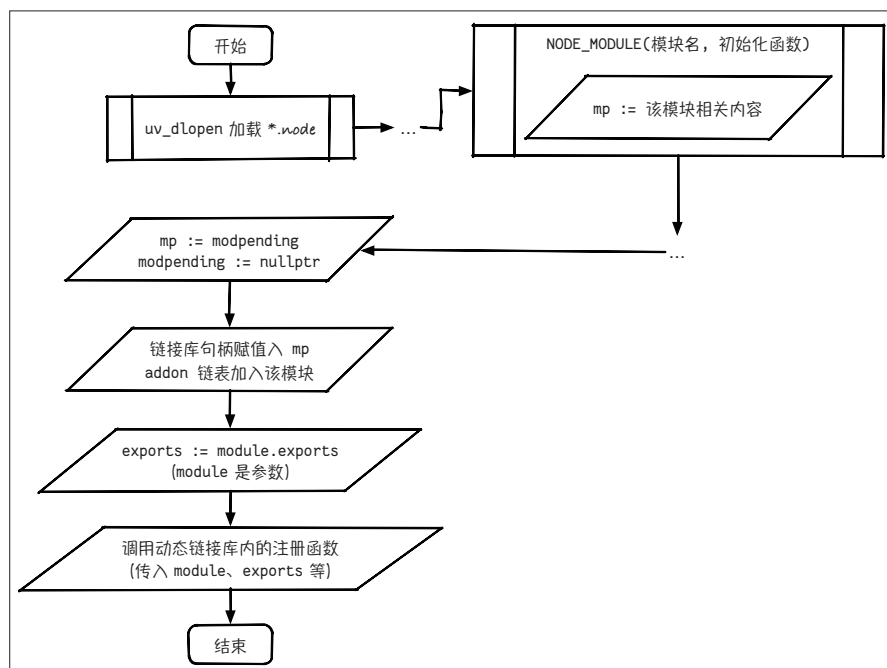


图 2-7 DLOpen 流程图

至此，几种 Node.js 模块的最后一种 C++ 扩展模块加载原理也解析完成了。

可能有些读者有疑问：怎么保证下次用到这个 modpending 的时候它是当前加载注册的模块，而不会被别的模块覆盖呢？

Node.js 在主事件循环中的所有操作都是单线程的，而一个按照正常 Node.js 规范写的代码或者模块当然也是这样的。这样 require 函数加载一个 C++ 模块的时候也是单线程的。所以，在加载 Node.js 模块的时候就不存在资源抢占和锁的问题。

2.2.3 小结

本节首先介绍了 C++ 扩展模块的本质，实际上就是一个对应各操作系统的动态链接库，只不过暴露出了特定的 API。

然后介绍了 4 种不同类型的 Node.js 模块的加载原理。

其中 C++ 核心模块会通过 `NODE_MODULE_CONTEXT_AWARE_BUILTIN` 等宏将不同的模块注册进 Node.js C++ 核心模块链表中；而 Node.js 内置模块则会在 Node.js 编译时被写入 C++ 源码中并被编译到 Node.js 可执行二进制文件中，并在恰当的时机被拿出来闭包化导出；用户源码模块会在首次执行 `require` 的时候被读取源码并闭包化导出，然后再加入模块缓存中；C++ 扩展则会在首次执行 `require` 的时候通过 `uv_dlopen` 加载该扩展的 *.node 动态链接库文件，在链接库内部把模块注册函数赋值给 `modpending`，然后将执行 `require` 时传入的 `module` 和 `exports` 两个对象传入模块注册函数进行导出。

至此，希望读者对 Node.js 的模块原理尤其是其 C++ 扩展模块的原理有一个更深层次的理解。

2.2.4 参考资料

- [1] C/C++ Addons: https://nodejs.org/docs/v6.9.4/api/addons.html#addons_addons.
- [2] NodeJS 源码详解——process 对象: <http://blog.hellofe.com/nodejs/2013/10/24/Learn-Node-Source-Code-One/>.
- [3] 朴灵. 深入浅出 Node.js[M]. 北京: 人民邮电出版社, 2013. 20-27.
- [4] 详解 NodeJs 的 VM 模块: <http://www.alloyteam.com/2015/04/xiang-jie-nodejs-di-vm-mo-kuai/>.

3

Chrome V8 基础

在第 1 章和第 2 章中，读者了解了关于 Node.js 模块和 C++ 扩展的基本原理，从而得知 Node.js 是基于 Chrome V8（下面简称 V8）引擎进行开发的。

所以，自然而然在进行 Node.js 的 C++ 扩展时，读者会需要了解一定的 Chrome V8 基础知识，包括一些原理和 API 等，这样才能更好地进行开发。

如果大家想快速开发一个 C++ 扩展的样例以保持新鲜感和成就感，可以先跳过本章，进行实战演练，之后再回过头来阅读本章。

为了本书的一致性，本章所讲解的 V8 是 Node.js v6.9.4 所对应的 V8 版本。在兼容性允许的情况下，读者可自行选择其他版本。

3.1 Node.js 与 Chrome V8

在 1.3.1 节中曾提到，Node.js 的 JavaScript 运行时引擎是 Chrome V8，那么它们究竟是以怎样的形式连接起来的呢？

我们先回顾一下历史。

- 2006 年，V8 开始投入研发。
- 2008 年 9 月，V8 发布了第一个版本，并且 Chrome 浏览器也几乎同期发布了。

至此，JavaScript 还是运行在浏览器中的一门脚本语言。

- 2009 年，Ryan Dahl 开发了 Node.js。

之后 JavaScript 就被带到了后端领域中，V8 也不仅仅只是 Chrome 的一个支持引擎了。

与 Chrome 差不多，Node.js 为 V8 的运行提供了一个宿主，只不过前者宿主中包含的是类似于 HTML DOM、window 对象等内容；而后者则是提供了一整个沙盒（vm）机制，以及文件系统、网络操作等内容。

也就是说，Node.js 实际上就是 Chrome V8 引擎的一个宿主。如果你有兴趣，也完全可以用 Chrome V8 创建一个别的 .js，比如 Mode.js、Lode.js 等。

基于上面的一个说法，其实我们不需要对 Chrome V8 有一个特别深入的了解，也不需要知道它的算法、原理等。我们关心的只是它暴露出来的一些 API，以及使用这些 API 所必要储备的知识。

有了这些 API，我们就能让自己的 C++ 扩展与 Node.js 进行互通了——因为 Node.js 底层很大程度上也是直接使用了 Chrome V8 所暴露出来的 API。

值得注意的是，本章介绍的一些 Chrome V8 知识是浅尝辄止的，如果大家对 Chrome V8 有更深层次的兴趣，可以访问 <https://developers.google.com/v8/> 来获取更多信息。

3.2 基本概念

3.2.1 内存机制

在 Chrome V8 中，内存机制是非常重要的，其中就包括它内在的各种概念。V8 高效的一个重要原因就是它的内存机制，这些概念需要事先系统性地了解，否则在实际开发中很容易就会踩“坑”。

Chrome V8 中 JavaScript 的数据类型都是由 V8 内部的内存机制进行管理的。

为了读者能形成更清晰的概念，这里有必要再强调一下 Node.js 底层需要完全理解的点。

Node.js 实际上就是一个使用 C++ 完成的程序，其能执行 JavaScript 代码，它的底层主要由两部分第三方库组成——Chrome V8 和 libuv：

- Chrome V8 是 JavaScript 运行时，用于解释执行 JavaScript。
- libuv 实现了 Node.js 中大家老生常谈的“事件循环”。

其中 Chrome V8 是一个由 C++ 完成的库，用于执行 JavaScript。也就是说如果你在自己的 JavaScript 代码中声明了一个变量，那么这个变量将被 V8 的内存机制进行管理。

Chrome V8 所创建的 C++ 数据类型能被你编写的 C++ 代码（如 C++ 扩展）所访问，并且其跟 JavaScript 中操作的是在内存中相同的存储单元。也就是说你在 JavaScript 中声明了一个 `let a = 1;`，那么在相应的作用域下，你的 C++ 代码能对其进行操作。

笔者前面所说的 Chrome V8 创建的数据存储单元只能被它的内存回收机制所回收，而不能被我们自己进行管理（不能被 `delete` 或者 `free` 等操作符操作）。而回收的前提是——这个 JavaScript 变量在 JavaScript 代码中已经不再被引用了，而且在你的 C++ 代码中也不再被引用了。

这里笔者给大家稍稍展示一下原生 C++ 的数据类型和 Chrome V8 中数据类型的区别。

在 C / C++ 中，我们通常最开始学的字符串类型就是一个以 `'\0'` 结尾的字符数组，一般以 `char*` 或者 `char[]` 的形式存在。

而在 JavaScript 中，一个类似于 `let a = "hello world"` 这样的字符串，Chrome V8 会新建一个 `v8::String` 类型的对象，里面有一堆 Chrome V8 内置的数据信息。

类似的还有浮点数在 C++ 中通常是 `float` 或者 `double`；而在 Chrome V8 中则是 `v8::Number`，这也是一个对象。

这些数据类型中都有对数据元信息的一个引用，用于内存管理。

作为一个 Node.js 开发者，大家可能对于老生代内存、新生代内存等耳熟能详。实际上 Chrome V8 中的堆内存可不止这两个部分。

- **新生代内存区：**基本的数据对象都被分配在这里，其特点是小而频。其区域小但是垃圾回收频繁。
- **老生代指针区：**这是一堆指向老生代内存区具体数据内容的指针。基本上从新生代蜕变过来的对象会被移动至此。
- **老生代数据区：**存放数据对象，而不是指向其他对象的指针。老生代指针区的指针就往这边指。
- **大对象区：**这里存放体积超越其他区大小的对象，每个对象有自己的内存，垃圾回收并不会移动大对象。
- **代码区：**代码对象，也就是包含 JIT 之后指令的对象，会被分配在这里。这是唯一拥有执行权限的内存区。
- **Cell 区、属性 Cell 区、Map 区：**存放 Cell、属性 Cell 和 Map，每个区域都是存放相同大小的元素，结构简单。

1. 新生代内存

绝大多数 JavaScript 对象都会被分配到新生代内存中，内存区域很小但是垃圾回收频繁。

在新生代分配内存非常容易，我们只需要保存一个指向内存区的指针，不断根据新对象的大小进行递增即可。当该指针到达了新生代内存区的末尾时，就会有一次清理。

新生代内存使用 Scavenge 算法进行回收，其具体思想可以参阅网络中的一些文献。

该种算法中的大致思想为，将内存一分为二，每部分的空间都被称为 Semispace。在两个 Semispace 中，总有一个处于使用状态，称为 From 空间；另一个处于闲置状态，称为 To 空间。

在分配对象时，总使用 From 空间进行分配；在垃圾回收时，Chrome V8 检查 From 空间中的存活对象，然后将这些对象复制到 To 空间中，剩下的对象就会被释放，完成复制后 From 空间和 To 空间的角色对调，原来的 From 空间变成了新的 To 空间，而原来的 To 空间就变成了 From 空间。

在此可以看出，新生代内存中，总有至少一半的内存是空闲不用的。不过所幸的是正如前面所述，新生代内存的特点是空间小、垃圾回收频繁，所以也浪费不了多少空间。

而当一个新生代中的对象经过多次新生代的垃圾回收而继续坚挺在内存区中时，则说明它不是一个“炮灰”，其生命周期较长，是一个“将相之才”。这个时候它就会被移动到老生代内存。这种操作被称为**对象的晋升**。

晋升的标准有两条：

- 在垃圾回收的过程中，如果该对象已经经历过一次新生代的清理，那就会晋升（这类似于游戏晋级赛，经过一次筛选后没被淘汰，这样的选手就晋级了）。
- 在垃圾回收的过程中，如果其中 To 空间的使用已经超过了 25%，那么这个对象也会晋升（这类似于游戏补位晋级，如果淘汰区的位置已经满了，那么当前选手不用被筛选也能直接晋级）。

2. 老生代内存

老生代所保存的对象大多数是生存周期很长的甚至是常驻内存的对象，而且老生代占用的内存较多。

Chrome V8 的特点就在于，无论是 Chrome 还是 Node.js，其老生代内存占用的空间都是非常大的。如果这里再使用 Scavenge 算法进行垃圾回收，那浪费的内存就更大了。

于是，它们的垃圾回收机制就变成了 Mark-Sweep 和 Mark-Compact 的结合体。其主要采用 Mark-Sweep。如果老生代空间不足以分配从新生代晋升过来的对象时，才使用 Mark-Compact。

（1）Mark-Sweep（标记清除）

标记清除其实是两个词——“标记”和“清除”。

- **标记：**在标记阶段需要遍历老生代堆中的所有对象，并标记那些活着的对象，然后进入清除阶段。
- **清除：**在清除阶段，Chrome V8 只清除没有被标记的对象。

由于标记清除只清除死亡对象，而死亡对象在老生代中占用的比例通常较小，因此其效率还是比较高的。

可以想象你在一筐苹果中挑出两三个烂苹果，还是很快的，至少比从一筐苹果中挑出半筐烂苹果要快得多。

（2）Mark-Compact（标记整理）

在标记清除时，容易产生内存碎片的问题。所以 Mark-Compact 在标记清除的基础上进行修改，在清除的时候让它们变得紧缩。

这相当于在清除的时候，让活着的剩余对象尽可能往内存区域的前面靠，直到内存区域前排全部排满，而后部区域是空的。

标记整理的过程涉及内存区域的紧缩，所以其效率其实比标记清除要低。不过其优势是不会产生内存碎片。

（3）惰性清理

在标记时，哪些对象是死的，哪些对象是活的，Chrome V8 已经掌握了。但是清理释放是需要开销的，所以 Chrome V8 并不急着去清理，而是延迟进行。垃圾回收器可以根据自身需要来清理死掉的对象。

3.2.2 隔离实例（Isolate）

在 Chrome V8 中，一个引擎实例的数据类型叫 Isolate。很多人可能有疑问，为什么实例不是 Instance 而是 Isolate¹ 呢？

事实上它的全称是隔离实例（Isolated Instance），意为与别的 Isolated Instance 实例完全隔开的一个实例，互不干扰。

¹ Isolate 的意思是隔离。

这是 Chrome V8 中所有要执行的地方都要出现的数据。它就是一个 V8 引擎的实例，也可以理解为引擎本体。每个实例内部拥有完全独立的各种状态，包括堆管理、垃圾回收等。

通过一个实例生成的任何对象都不能在另一个实例中使用。什么意思呢？抛开 Node.js 不说，使用 Chrome V8 进行开发的开发者其实是在它的程序中创建多个 Isolate 实例，并且并行地在多个线程中使用的——但是一个实例不能在多线程中使用。

但是实例自身并不执行 JavaScript，也没有 JavaScript 环境里面的上下文。

我们能通过下面的代码创建一个实例：

```
// 省略 V8 初始化过程

// 实例所必要的参数
v8::Isolate::CreateParams create_params;

// 省略参数设置过程

// 创建一个实例
v8::Isolate* isolate = v8::Isolate::New(create_params);
```

不过值得注意的是，在我们开发 Node.js 的 C++ 扩展时，我们已经处于 Chrome V8 的环境中，这时就不需要再生成一个实例了，直接获取 Node.js 环境所使用的实例即可。

如在 Node.js v6.9.4 中，我们在扩展函数中应该这么写：

```
void Method(const v8::FunctionCallbackInfo<v8::Value>& args)
{
    Isolate* isolate = args.GetIsolate();

    // ...
}
```

3.2.3 上下文 (Context)¹

上下文对象是用来定义 JavaScript 执行环境的一个对象，其数据类型是 Context，它在创建的时候要指明属于哪个实例。

```
v8::Isolate* isolate = ...;
v8::Local<v8::Context> context = v8::Context::New(isolate);
```

¹ 关于上下文的详细内容会在 3.5 节继续展开。

这里可以理解为它是一个沙箱化的执行上下文环境，内部预置了一系列的对象和函数。

有关上下文的内容并不会详述，毕竟在 Node.js 的 C++ 扩展中，通常我们不需要用一个上下文来执行。

3.2.4 脚本（Script）

脚本（Script）的概念顾名思义，就是一个包含一段已经编译好的 JavaScript 脚本的对象，数据类型就是 Script。它在编译时就与一个处于活动状态的上下文进行绑定。

```
v8::Local<v8::Context> context = ...;

...

v8::Local<v8::String> source = 一段 JavaScript 代码;

// 与上下文绑定并编译
v8::Local<v8::Value> result = v8::Script::Compile(context, source).
ToLocalChecked();

// 执行脚本
v8::Local<v8::Value> result = script->Run(context).ToLocalChecked();

...
```

3.2.5 小结

本节介绍了 Chrome V8 开发的一些基础知识，包括它的内存机制是怎样的，以及它的数据类型实际上不是原生 C++ 的数据类型（如 double、int 等）。

另外，Chrome V8 有着一套强大的内存管理机制。本节浅述了以空间换时间的高效新生代内存回收算法 Scavenge，以及效率与碎片管理并存的 Mark-Sweep 和 Mark-Compact 结合体机制下的老生代内存回收算法。同时也介绍了新生代与老生代内存的一个联系，在特定情况下，新生代内存里面的对象会晋升到老生代对象中。

在介绍完 Chrome V8 的内存管理机制之后，本节还简单介绍了 Chrome V8 的几个基础类型的数据结构，分别是引擎实例 Isolate、沙箱化可执行上下文 Context 和脚本 Script。

这些基础知识会对读者阅读后续章节产生帮助。

3.2.6 参考资料

- [1] Scott Frees. C++ and Node.js Integration[EB/OL]: <https://scottfrees.com/ebooks/nodecpp/>.
- [2] A tour of V8: Garbage Collection: <http://www.jayconrod.com/posts/55/a-tour-of-v8-garbage-collection>.
- [3] 浅谈 V8 引擎中的垃圾回收机制: <http://blog.duckbill-ideas.com/20140810/%E6%B5%85%E8%B0%88%E5%BC%95%E6%93%8E%E4%B8%AD%E7%9A%84%E5%9E%83%E5%9C%BE%E5%9B%9E%E6%94%B6%E6%9C%BA%E5%88%B6/>.
- [4] 朴灵. 深入浅出 Node.js[M]. 北京: 人民邮电出版社, 2013. 111-136.
- [5] What exactly is the difference between v8::Isolate and v8::Context?: <http://stackoverflow.com/questions/19383724/what-exactly-is-the-difference-between-v8isolate-and-v8context>.
- [6] V8 API Reference Guide: <https://v8docs.nodesource.com/node-6.12/index.html>. 若读者打开该地址, 却发现页面不存在, 可直接前往 <https://v8docs.nodesource.com/>, 并点击“6.x”字样的超链接进入(注意该地址经常换)。

3.3 句柄 (Handle)

精通 Windows C++ 开发的读者可能对句柄 (Handle) 这个概念并不陌生。的确 V8 的句柄和 Windows C++ 开发下的句柄有一定的相似之处。

句柄是 Chrome V8 中的一个重要概念, 它提供了对于堆内存中 JavaScript 数据对象的一个引用。

问: 为什么使用句柄, 在内部提供引用, 而不直接使用一个对象指针之类的东西呢?

答: Chrome V8 在进行垃圾回收的时候, 通常会将 JavaScript 的数据对象移来移去。如果使用指针的话, 一旦一个对象被移走, 这个指针就成了野指针。而在移动的过程中, 垃圾回收器会更新引用了这个数据块的那些句柄, 让其断不了联系。

拟人:

- 指针形式: 老司机 (对象) 家住五楼, 你有一个传送门 (指针) 就指向了五楼, 这样你就能直接找到老司机。但是突然有一天老司机搬家了, 搬进来一个新司机 (不知道哪里来的野对象, 可能合法, 也可能非法)。你再进传送门, 就找不到老司机了。

- 句柄形式：老司机（对象）家住五楼，你有一个传送门在传送中心（垃圾回收器）登记了，说明要传送到老司机的家。突然有一天传送中心让老司机搬家了，并更新了所有指向老司机家的传送门。你再进传送门，仍然能找到老司机。

当一个对象不再被句柄引用时，那么它将很不幸地被认为是垃圾。Chrome V8 的垃圾回收机制会不时地对其进行回收。所以，句柄的引用对于 Chrome V8 的垃圾回收是很关键的¹。

话说回来，句柄在 Chrome V8 中只是一个统称，它其实还分为多种类型：

- 本地句柄（`v8::Local`）；
- 持久句柄（`v8::Persistent`）；
- 永生句柄（`v8::Eternal`）；
- 待实本地句柄；
- 其他句柄。

其中，本地句柄和持久句柄是在 Chrome V8 以及 Node.js 中最常用到的句柄，而永生句柄和其他类型的一些句柄比较罕见。

句柄存在的形式是 C++ 的一个模板类，其需要根据不同的 Chrome V8 数据类型进行不同的声明。例如：

- `v8::Local<v8::Number>`：本地 JavaScript 数值类型句柄。
- `v8::Persistent<v8::String>`：持久 JavaScript 字符串类型句柄。

这些句柄类都能通过 `.` 方法名 来访问句柄对象的一些方法。而它还重载了 `*` 和 `->` 两个操作符。通过 `*` 操作符能得到这个句柄所引用的 JavaScript 数据对象实体指针，`->` 也一样。

假设我们有一个字符串本地句柄 `Local<String> str`，那么就可以有这样的一些调用：

- `str.IsEmpty()`：句柄对象本身的函数，判断这个句柄是否是空句柄。
- `str->Length()`：通过 `->` 得到 `String*`，而 `String` 有一个方法 `Length` 可获取字符串长度，所以 `str->Length()` 是这个句柄所指的字符串实体的长度。

3.3.1 本地句柄（Local）

本地句柄存在于栈内存中，并在对应的析构函数被调用时被删除。比较悲惨的是，它们的生命周期是由其所存在的句柄作用域（Handle Scope）决定的。

¹ 更深的内容本书不进行讲解，有兴趣的读者可以阅读“V8 Design Elements”来获取更多有价值的知识：<https://github.com/v8/v8/wiki/Design-Elements>。

通常情况下，一个句柄作用域对象会在一个函数体内的一开始被声明。当一个句柄作用域对象被删除的时候，之前在这个句柄作用域内创建的那些句柄所指的对象如果没有别的地方引用的话，就可以被垃圾回收器自由释放了。

而根据 C++ 的对象栈生命周期，这个句柄作用域对象会在函数结束的时候被删除。

关于句柄作用域的内容还会在下面继续解释。

注意：句柄栈并不是 C++ 调用栈的子集，但是句柄作用域对象却是。并且句柄作用域只能在栈内存中被分配——也就是说不能用 `new` 创建一个句柄作用域¹。

对于本地句柄来说，读者需要熟知几个比较重要的 API。

1. 创建 (New)

`New` 是 `Local` 句柄的一个静态方法。在 Node.js v6.9.4 对应的 V8 版本中有两个重载。

- `Local<T>::New(Isolate* isolate, Local<T> that)`：传入 `Isolate` 实例和另一个本地句柄，进行复制构造（`T` 代表 C++ 模板类中的 V8 JavaScript 数据类型）。
- `Local<T>::New(Isolate *isolate, const PersistentBase<T> &that)`：传入 `Isolate` 实例和一个持久句柄。

除此之外，本地句柄还有一堆操作符重载，如 `==`、`->` 等。更多的 API 可以直接参考它的文档(https://v8docs.nodesource.com/node-6.12/de/deb/classv8_1_1_local.html。若读者打开该地址，却发现页面不存在，可直接前往 <https://v8docs.nodesource.com/>，并点击“6.x”字样的超链接进入，注意该地址经常换）。

另外，大多数时候，我们会通过 Chrome V8 中的 JavaScript 数据类的一些静态方法来获得一个本地句柄，例如：

```
Local<Number> Number::New(Isolate* isolate, double value);
```

就是一个 `v8::Number` 类型的静态方法，其传入一个 `Isolate` 实例以及一个 C++ 的 `double` 类型数据，Chrome V8 就会创建一个 JavaScript 中的 `Number` 数据，然后返回一个指向内存中引用该数据的本地句柄。

其他数据类型的获得本地句柄的函数可参考 Chrome V8 的类参考文档（<https://v8docs.nodesource.com/node-6.12/d2/dc3/namespacev8.html>。若无法打开，则请参照本节前面所述的方法打开），或者阅读本书的后续章节获得。

¹ 该句原话源于 Chrome V8 的 *Embedder's Guide*：<https://github.com/v8/v8/wiki/Embedder%27s-Guide>。

值得注意的是，书中的这些 API 参考均针对 Node.js v6.9.4 所用的 V8 版本（5.1.281）及其上下的一些版本。这里不保证所有版本的兼容性。如果需要获得最新的其他版本 Chrome V8 的精确 API 情况，请参照各自版本的 Chrome V8 文档。本书主要讲解 Chrome V8 的一些概念和思想。

2. 清除 (Clear)

将这个句柄设置成为一个空句柄——也就是将其指向“空”。

尽管不是很准确，也不论代码是否会造成内存泄漏，但可以将 Clear 函数类比成这样的操作：

```
int* p = new int(0);

// 类比于 Local<T>::Clear()
p = NULL;
```

举一个 Clear 的例子：

```
Local<Number> handle = Number::New(isolate, 233);
handle.Clear();
```

3. 是否为空 (IsEmpty)

如果一个指针是空指针，我们通常用 == 来判断：

```
// 假设有一个 int* 的指针 p
if(p == NULL)
{
    // ...
}
```

如何判断一个 Chrome V8 的本地句柄是不是一个空句柄，并不是直接用 == 来判断，而是用 IsEmpty 函数来判断。

```
Local<Number> handle = Number::New(isolate, 233);

// 只是举例说明 `IsEmpty` 的用法
// 不用太较真它在这个例子中的什么时候是空的
if(handle.IsEmpty())
{
    // ...
}
```

当一个句柄是空句柄的时候，我们对其直接进行一些操作会导致程序崩溃，这类似于一个指针是 `NULL` 但是我们还是去调用这个对象的一些方法。为此 Chrome V8 推出了一种 `MaybeLocal` 的待实本地句柄，可以阅读 3.3.4 节获得更多相关信息。

4. 转换数据类型 (As / Cast)

将某种数据类型的本地句柄转换成另一种类型的本地句柄，就可以用 `As` 或者 `Cast` 函数了，其中 `As` 是成员函数，而 `Cast` 是静态函数。

比如我们拿到了一个 `Local<Value>` 的本地句柄 `handle`，我们就能通过这两个函数将其转换为其他类型的本地句柄，如 `Number`：

```
Local<Number> ret1 = Local<Number>::Cast(handle);
Local<Number> ret2 = handle.As<Number>();
```

3.3.2 持久句柄 (Persistent)

持久句柄提供了一个堆内存中声明的 JavaScript 对象的引用。持久句柄与本地句柄在生命周期上的管理是两种不同的方式。

当你认为世界那么大，一个 JavaScript 对象不应该只存在于当前的句柄作用域中，而应该出去看看的时候，就应该对这个 JavaScript 对象使用持久句柄。

举一个简单的例子，Google Chrome 中的 DOM 节点们在 Chrome V8 中就是以持久句柄的形式存在的——它们不局限于某个函数的作用域中。

持久句柄可以变成“战五渣”¹。我们能通过使用 `PersistentBase::SetWeak` 来使一个持久句柄变弱 (Weak)，成为一个弱持久句柄。

当对一个 JavaScript 对象的引用只剩下一个弱持久句柄时，Chrome V8 的垃圾回收器就会触发一个回调。

除弱持久句柄以外，持久句柄还分唯一持久句柄 (`v8::UniquePersistent<...>`) 和一般持久句柄 (`v8::Persistent<...>`)。

- 唯一持久句柄使用 C++ 的构造函数和析构函数来管理其底层对象的生命周期。
- 一般持久句柄可以使用它的构造函数来进行创建，但是必须调用 `Persistent::Reset` 来进行显式地清除。

¹ 完整句子为“战斗力为五的渣渣”，最早说出这句话的人是漫画《七龙珠》中的人物拉蒂兹。其大意为很弱。

如无特殊说明，笔者通常说的持久句柄指的是一般持久句柄。

谈一个问题

无论是唯一持久对象还是一般持久对象，都无法被复制。也就是说在 C++ 11 之前，它们都无法作为标准库中容器的值来使用。

不过 Chrome V8 提供了 `v8::PpersistentValueMap` 和 `v8::PersistentValue-Vector` 两个容器，其用起来类似于 STL 中的 `map<...>` 和 `vector<...>`，让你可以在持久句柄的容器海洋中自由徜徉。

前面介绍了这个问题是在 C++ 11 之前才会出现的。在 C++ 11 中其实并不需要 `v8::PpersistentValueMap` 了，因为它解决了这个问题。

持久句柄继承自持久句柄的基类 (`PersistentBase<T>`)，它有一些我们可能会用到的 API。对于本书未提及的 API，建议读者阅读 Chrome V8 文档的 `Persistent` (https://v8docs.nodesource.com/node-6.12/d2/d78/classv8_1_1_persistent.html) 与 `PersistentBase` (https://v8docs.nodesource.com/node-6.12/d4/dca/classv8_1_1_persistent_base.html)。若文档无法打开，则请参照前面所述的方法进行打开。

1. 构造函数

与本地句柄不同的是，持久句柄通常是通过本地句柄升格而成。所以它的获得方法通常是在构造函数中传入一个本地句柄。

持久句柄的构造函数有几种常用的重载。

- `Persistent()`：直接创建一个持久句柄，这种方法获得的持久句柄通常会在后续再调用别的方法对一个本地句柄进行升格。
- `Persistent(Isolate *isolate, Local<T> that)`：传入 `Isolate` 实例以及一个本地句柄，能得到这个本地句柄所引用的 Chrome V8 数据对象的一个持久句柄，例如：

```
Local<Number> local = Number::New(isolate, 2333);
Persistent<Number> persistent_handle(isolate, local);
```

- 其他构造函数请参考 Chrome V8 的文档 (https://v8docs.nodesource.com/node-6.12/d2/d78/classv8_1_1_persistent.html。若文档无法打开，则请参照前面所述的方法进行打开)。

2. 清除 (Clear) 与是否为空 (IsEmpty)

与本地句柄一样，持久句柄一样有这两个函数。只不过它的这两个函数是从 `PersistentBase` 继承而来的——但是你一样可以放心使用它们。

3. 置为弱持久句柄 (SetWeak)

这是持久句柄中的一个重要函数，按照字面意思解读就是，将这个持久句柄降格为一个弱持久句柄。

下面先来看一下弱持久句柄的函数原型。

```
template <typename P>
void Persistent<T>::SetWeak(P *parameter, WeakCallbackInfo<P>::Callback
callback, WeakCallbackType type);
```

其中 P 可以是任意数据类型。这是一个比较有意思的原型，笔者逐个来讲解它的 3 个参数。

- parameter: 可以是任意数据类型，用于第二个参数。
- callback: 一个回调函数。笔者在先前介绍过，当对一个 JavaScript 对象的引用只剩下一个弱持久句柄时，Chrome V8 的垃圾回收器就会触发一个回调函数。那里所说的回调函数就是这个回调函数了。触发这个回调函数的时候，第一个参数 parameter 会被传入回调函数供你使用。
- type: 回调函数的类型，是一个枚举，有以下两个值。
 - kParameter: 使用该值时，在回调函数中用 WeakCallbackInfo::GetParameter() 获取传入的数据。
 - kInternalFields: 使用该值的时候，在回调函数中用 WeakCallbackInfo::GetInternalField(int index) 等获取数据。关于 InternalField 的内容将会在后续章节中介绍。

到此为止，我们还差一个内容未介绍，那就是 WeakCallbackInfo<P>::Callback 回调函数。其在 Chrome V8 的文档中同样曾提及。

```
typedef void(*Callback)(const WeakCallbackInfo<T> &data);
```

讲得通俗一点，如果你有一个类是 Test，要将它作为回调参数的话，回调函数就应该这么写：

```
void Callback(const WeakCallbackInfo<Test>& data)
{
    // 做回调的内容，比如删掉你的 `Test` 对象指针以免泄漏。
    // 因为这个回调函数一旦被触发，就代表再也没有其他本地句柄、持久引用等
    // 任何句柄指向它了。
    Test* test = data.GetParameter();
```

```
delete test;
}
```

有了这个回调函数之后，就能将它传入 `SetWeak` 函数中。

```
// 没有任何其他地方删除它
Test* test = new Test();
// 假设 `persistent_handle` 是任意一个持久句柄，可能是 `String`，也可能是 `Number`
// 等——无关紧要
persistent_handle.SetWeak(test, Callback, WeakCallbackType::kParameter);
```

笔者现在再来解释一下上面两句代码的含义。在 `persistent_handle` 快要被垃圾回收的时候（通常我们可以认为“快要被垃圾回收的时候”即“一个数据对象只剩下一个弱持久句柄的时候”，除非开发者有别的特殊用法），就将一个 `test` 对象传入 `Callback` 函数；而在 `Callback` 函数中，程序将会把 `test` 对象删除。

大家再仔细想想，本节是否还遗漏了什么内容？那就是 `WeakCallbackInfo` 到底是什么，里面又写了什么内容，要在回调函数中怎么用。下面笔者就再讲解一下 `WeakCallbackInfo` 吧。首先它是一个模板类。模板中的类型可以是任意类型，总之跟前面介绍的 `parameter` 一致即可，毕竟我们要在回调函数中通过这个 `WeakCallbackInfo` 来获取传进来的 `parameter`。

然后这里再列举 `WeakCallbackInfo` 的几个重要 API。

- `Isolate* GetIsolate()`：获取 `Isolate` 实例。
- `P* GetParameter()`：获取 `SetWeak` 时传入的第一个参数。
- `void* GetInternalField(int index)`：根据传入的索引值返回对应持久句柄的数据对象 `InternalField` 的指定值，以 `void*` 无类型的指针形式返回，使用时通常需要自行进行类型转换（读者可以认为只有 `Object` 对象的持久句柄能够使用该函数，其他类型或者不能使用，或者一般情况下用不到）。

4. 取消弱持久句柄（`ClearWeak`）

既然有 `SetWeak` 函数，那么 `Chrome V8` 也相应地有 `ClearWeak` 函数。其作用就是取消它的弱持久句柄——又变成一般的持久句柄了。

该函数没有参数：

```
persistent_handle.ClearWeak();
```

5. 标记独立句柄 (MarkIndependent)

将一个持久句柄标记为独立的 (Independent)。

独立的持久句柄有两个特性：

- ① Chrome V8 的垃圾回收器可以自由地忽略包含这个句柄的对象组¹。
- ② 独立的持久句柄可以在新生代回收的时候被回收，而非独立句柄则不行。若是通俗、不精确地描述，则可以说独立持久句柄的生命周期更短。

在 Node.js 的 C++ 扩展中，我们应该只用到它的第二个特性，并且通常与 SetWeak 成对出现，因为我们通常期望弱持久句柄更“短命”。

```
persistent_handle.SetWeak(test, Callback, WeakCallbackType::kParameter);
persistent_handle.MarkIndependence();
```

6. 是否为弱的 (IsWeak) 与是否为独立的 (IsIndependent)

顾名思义，这两个函数用于判断一个持久对象是否是弱的或者独立的。

```
bool weak = persistent_handle.IsWeak();
bool independent = persistent_handle.IsIndependent();
```

3.3.3 永生句柄 (Eternal)

我们认为这种句柄在程序的整个生命周期内是不会被删除的。比起持久句柄来说，永生句柄的开销更小。

问：为什么永生句柄的开销更小？

嘚瑟地答：因为它不需要垃圾回收啊。

这种句柄在 Node.js 的 C++ 扩展实际开发中通常用不到，这里就不再赘述了。有兴趣的读者可以直接阅读 Chrome V8 的 API 文档。

3.3.4 待实本地句柄 (Maybe Local)

待实本地句柄是新版(自 2015 年 2 月 27 日 4.3.10 版本后)Chrome V8 中另一种重要的句柄。

读者先看一下一种旧版 Chrome V8 下写的代码：

¹ 目前 Node.js 未使用对象组的概念，所以不必深究。它应该是为 Chromium 中的 DOM 节点准备的。

```
Local<Value> x = some_value;
Local<String> s = x.ToString();
s->Anything();
```

且不论这段代码本身是否有问题，在 `ToString()` 函数的内部如果本身发生了异常的话，`s` 将会是一个空的本地句柄。这个时候执行 `s->Anything()` 就会导致程序崩溃。

所以，在这种情况下我们需要进行一个判断才能保证程序的健壮性：

```
Local<Value> x = some_value;
Local<String> s = x.ToString();

if(!s.IsEmpty()) {
    s->Anything();
}
```

但显然，如果我们每次在使用一个句柄前都要做这么一个判断的话，就大大增加了系统的复杂度。而且，实际上有些数据类型的句柄并不需要检查 `IsEmpty`，那么开发者自己也经常不知道到底哪些返回值需要检查而哪些不需要检查了。

后来，**Chrome V8** 在新版中就推出了新的句柄——`MaybeLocal`，本书将这种句柄称为待实本地句柄，这意味着待落实它到底是不是一个有效的本地句柄。

在旧版的 `ToString` 函数中返回值是 `Local<String>`，而在这种句柄推出后，`ToString` 返回值就变成了 `MaybeLocal<String>`。

用一句话总结就是，那些在以往有可能返回空句柄的接口，现在都会以待实本地句柄的形式来代替返回值了。如果开发者需要拿到真正的本地句柄，就需要调用这个待实本地句柄的 `ToLocalChecked` 函数。例如：

```
MaybeLocal<String> s = x.ToString();
Local<String> _s = s.ToLocalChecked();
```

重要：`MaybeLocal` 的出现只是为了让你知道哪些地方的返回值需要检查是否为空，而不是确定一定不会返回空。在 `MaybeLocal` 的返回值下，开发者还要自行检测它是否为空，否则直接转换为 `Local` 还是会抛出异常。也就是说，除非是开发者非常确信上面的代码一定不会挂，否则仍要改成下面形式的代码：

```
MaybeLocal<String> s = x.ToString();

if(!s.IsEmpty())
{
    Local<String> _s = s.ToLocalChecked();
}
```


3.3.5 小结

本节介绍了 Chrome V8 开发中非常重要的两个概念——句柄和句柄作用域。

其中句柄是用于获取 JavaScript 对象实体的一种事物，有有效句柄连接的对象实体不会被垃圾回收器进行回收。而失去了所有句柄引用的对象实体被认为是垃圾，从而在下次垃圾回收的时候被释放。

在 Node.js 的 C++ 扩展开发中，我们通常会用到本地句柄、持久句柄和待实本地句柄 3 种类型的句柄，本节除了介绍了其各自的用处之外，还列举了几个重要的 API。

3.3.6 参考资料

- [1] Embedder's Guide: <https://github.com/v8/v8/wiki/Embedder%27s-Guide>.
- [2] V8/ChangeLog: <https://github.com/v8/v8/blob/master/ChangeLog>.
- [3] V8 里又有一大拨即将废弃的 API: <http://www.idom.me/articles/847.html>.
- [4] API changes upcoming to make writing exception safe code more easy: <https://groups.google.com/forum/#!topic/v8-users/gQVpp1HmbqM>.
- [5] v8::Persistent MarkIndependent, what does this method exactly do?: <http://stackoverflow.com/questions/16731246/v8persistent-markindependent-what-does-this-method-exactly-do>.

3.4 句柄作用域

若每次需要的时候均创建各式各样的本地句柄，一个函数写下来，你会发现本地句柄黏满了函数的各个角落——自己根本管不过来。

这个时候就轮到句柄作用域出场了。在代码中，句柄作用域以 `HandleScope` 或者 `EscapableHandleScope` 的形式存在于栈内存中，不要用 `new` 的形式去声明，否则就无法利用 C++ 作用域中对象生命周期的特性了。

正如前面所说的那样，句柄作用域实际上是一个维护一堆句柄的容器。当一个句柄作用域对象的析构函数被调用时，在这个作用域中创建的所有句柄都会被从栈中抹去。于是，通常情况下这些句柄所指的对象将会失去所有引用，然后会被垃圾回收器统一处理。

还有重要的一点是，作用域是一个套一个地以栈的形式存在的。在栈顶的句柄作用域处于激活状态。每次创建新的被管理对象的时候，都会将对象交付给栈顶的作用域管理，当栈顶作用域生命周期结束时，这段时间创建的对象就会被回收。

3.4.1 一般句柄作用域（Handle Scope）

笔者从 Chrome V8 的 “Getting Started” 中拉取一段样例代码进行讲解¹。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "include/libplatform/libplatform.h"
#include "include/v8.h"
using namespace v8;
class ArrayBufferAllocator : public v8::ArrayBuffer::Allocator {
public:
    virtual void* Allocate(size_t length) {
        void* data = AllocateUninitialized(length);
        return data == NULL ? data : memset(data, 0, length);
    }
    virtual void* AllocateUninitialized(size_t length) { return
    malloc(length); }
    virtual void Free(void* data, size_t) { free(data); }
};
int main(int argc, char* argv[]) {
    // 初始化 V8
    V8::InitializeICU();
    V8::InitializeExternalStartupData(argv[0]);
    Platform* platform = platform::CreateDefaultPlatform();
    V8::InitializePlatform(platform);
    V8::Initialize();
    // 创建一个新的 Isolate 对象并将其设为当前的 Isolate
    ArrayBufferAllocator allocator;
    Isolate::CreateParams create_params;
    create_params.array_buffer_allocator = &allocator;
    Isolate* isolate = Isolate::New(create_params);
    {
        Isolate::Scope isolate_scope(isolate);
        // 在栈内存中创建一个句柄作用域
        HandleScope handle_scope(isolate);
        // 创建一个 context
```

¹ 代码来源：<https://chromium.googlesource.com/v8/v8+/branch-heads/5.1/samples/hello-world.cc>，为 Chrome V8 文档中 “Getting Started” 的随文代码。

```

Local<Context> context = Context::New(isolate);
// 进入 context 以编译和执行 hello world 脚本
Context::Scope context_scope(context);
// 将一段 JavaScript 代码赋值给一个 V8 字符串，并得到句柄
Local<String> source =
    String::NewFromUtf8(isolate, "'Hello' + ', World!'",
                        NewStringType::kNormal).ToLocalChecked();

// 编译代码
Local<Script> script = Script::Compile(context, source).
    ToLocalChecked();
// 执行代码，得到结果
Local<Value> result = script->Run(context).ToLocalChecked();
// 将结果转换成 UTF8 字符串，然后打印
String::Utf8Value utf8(result);
printf("%s\n", *utf8);
}
// 妥善安排好 isolate 后事，然后关掉 V8
isolate->Dispose();
V8::Dispose();
V8::ShutdownPlatform();
delete platform;
return 0;
}

```

我们从 `Isolate::Scope isolate_scope(isolate);` 开始阅读，一直到打印结果。

其中 `Context::New()` 函数创建了一个 `Context` 对象，并返回这个对象的本地句柄，于是我们就能用 `Local<Context>` 类型的 `context` 来获取赋值了。

同理，下方的 `String` 等也与此类似。

我们不妨在 `Local<Context> context = Context::New(isolate);` 下加上一句代码，使其看起来像这样：

```

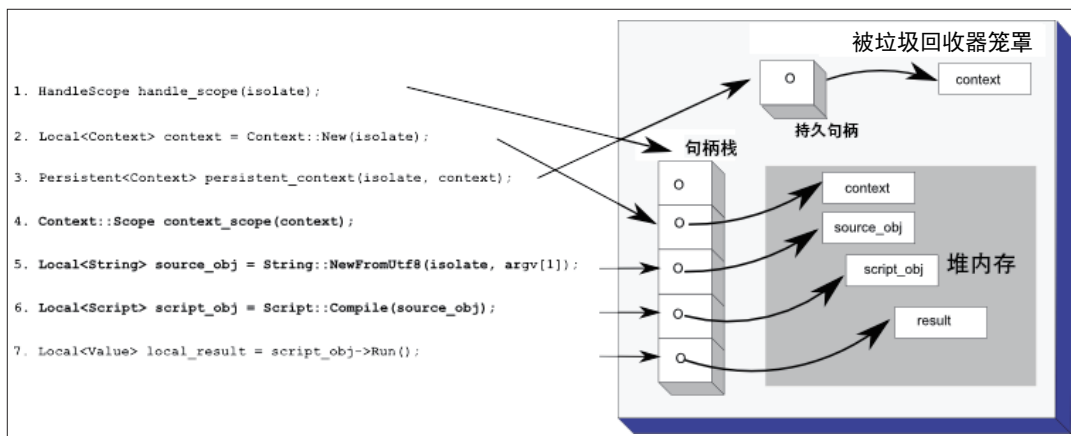
...

Local<Context> context = Context::New(isolate);
Persistent<Context> persistent_context(isolate, context);
Context::Scope context_scope(context);

...

```

然后对照图 3-1 开始介绍句柄与句柄作用域。

图 3-1 句柄与句柄作用域¹

- ① `HandleScope handle_scope(isolate);`: 创建一个句柄作用域，根据 C++ 的特性，在它所处的作用域结束时，其生命周期也就结束了，这个时候程序会自动调用它的构造函数来做一些事情（这种根据栈内存中对象生命周期特性来做一些事的思想还有 `ScopeLock`² 等）。
- ② `Local<Context> context = Context::New(isolate);`: 创建一个 `Context` 对象，并得到它的本地句柄——这个句柄存在于 `handle_scope` 的句柄栈中。也就是被这个句柄作用域对象所管理，以及它的真实对象存在于堆内存中，被垃圾回收器盯着。
- ③ `Persistent<Context> persistent_context(isolate, context);`: 基于 `context` 我们创建一个新的持久句柄和 `Context` 对象，它脱离了句柄作用域的掌控，直接受命于 `Chrome V8` 的垃圾回收器（这段代码在这里实际上也没什么用，加上这句话主要用于讲解持久句柄）。
- ④ `Context::Scope context_scope(context);`: 不进行讲解。
- ⑤ `Local<String> source = ...`: 可参照第二步解读，创建 `String` 对象并得到本地句柄。
- ⑥ `Local<Script> script = ...`: 可参照第二步解读。
- ⑦ `Local<Value> result = ...`: 可参照第二步解读。

最后，当 `HandleScope` 的析构函数被调用时，这些在这个句柄作用域中被创建的句柄和对象如果没有其他地方有引用的话，就会在下一次垃圾回收的时候被处理掉。

¹ 图片来源同样是 `Chrome V8` 的 *Embedder's Guide*。

² `ScopeLock` 是一种思想，主要用于线程中锁的自动释放。有兴趣的读者可以阅读笔者多年前的一篇博文，欢迎指正：<https://xcoder.in/2012/09/08/scope-lock/>。

不过还记得我们刚才加的那段持久句柄的代码吗？句柄作用域析构的时候并不会对它动手动脚。我们只能显式地调用 `Reset` 清除它。

3.4.2 可逃句柄作用域（Escapable Handle Scope）

通常在 JavaScript 代码中，一个函数返回一个值，这个值显然是一个本地句柄。

```
// 一个函数返回一个值
function returnValue() {
    let number = 2333;
    return number;
}
```

相似的代码如果要在 C++ 中用 Chrome V8 实现的话，也许是这样的：

```
v8::Local<v8::Number> ReturnValue()
{
    v8::Isolate* isolate = v8::Isolate::GetCurrent();
    v8::HandleScope scope(isolate);

    v8::Local<v8::Number> number = v8::Number::New(isolate, 2333);
    return number;
}
```

不过以上代码怎么看怎么别扭，总觉得哪里不对劲。

实际上在本地句柄中是有一个很明显的“坑”的。前面 C++ 代码中的 `number` 是一个本地句柄，归它的上层 `scope` 管。函数结束了，`scope` 的生命周期也就结束了，于是析构函数就被调用了。

这时，`number` 就被删除了，而其引用到的数值实体由于失去了引用也将被标记为垃圾——你再在这里返回，在外面使用，是不是不大合适呢？

想让 `number` 摆脱 `scope` 的“魔掌”吗？若想的话就该考虑考虑用可逃句柄作用域了。可逃句柄作用域的类型名是 `EscapableHandleScope`，它有一个 `Escape` 函数，可以给一个句柄以豁免权，将其复制到一个封闭的作用域中，并删除其他的本地句柄，然后返回这个新复制的句柄，即一个可以被安全返回的句柄。

结果之前的那个 `ReturnValue` 就应该被这样重写：

```
v8::Local<v8::Number> ReturnValue()
{
    v8::Isolate* isolate = v8::Isolate::GetCurrent();
```

```

v8::EscapableHandleScope scope(isolate);

v8::Local<v8::Number> number = v8::Number::New(isolate, 2333);
return scope.Escape(number);
}

```

如果大家看着这段代码觉得还不够过瘾，可以体验一把 Chrome V8 的 *Embedder's Guide* 上给出的代码：

```

// 这个函数返回一个包含 3 个元素 x、y、z 的数组
Local<Array> NewPointArray(int x, int y, int z) {
    v8::Isolate* isolate = v8::Isolate::GetCurrent();

    // 创建一个可逃句柄作用域
    EscapableHandleScope handle_scope(isolate);

    // 创建一个空数组
    Local<Array> array = Array::New(isolate, 3);

    // 如果数组创建错误，则直接返回一个空数组
    if (array.IsEmpty())
        return Local<Array>();

    // 填充数组
    array->Set(0, Integer::New(isolate, x));
    array->Set(1, Integer::New(isolate, y));
    array->Set(2, Integer::New(isolate, z));

    // 通过可逃句柄作用域返回数组
    return handle_scope.Escape(array);
}

```

解惑 1

我们来看看随书代码“2. cpp entry”的 entry.cpp 中的一段代码：

```

void RunCallback(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = Isolate::GetCurrent();
    HandleScope scope(isolate);

    Local<Function> cb = Local<Function>::Cast(args[0]);
    const unsigned argc = 1;
    Local<Value> argv[argc] = { String::NewFromUtf8(isolate, "hello
world") };
    cb->Call(isolate->GetCurrentContext()->Global(), argc, argv);
}

```

首先第一个问题是，`argv` 是一个 `Value` 的句柄数组，理应归 `HandleScope` 管理，为什么在最后能传入别的函数而不被回收呢？其实类似的问题还有 JavaScript 对象值返回时的代码：

```
// 把上述代码的最后一行 cb->Call(...) 替换成下面的代码
args.GetReturnValue().Set(argv[0]);
```

替换之后的代码表示，作为一个能被 JavaScript 调用的函数，这个函数在 JavaScript 的环境下返回 "hello world"。那么为什么这样写也是可以的呢？这里还是要强调生命周期，生命周期，生命周期。（重要的事情说3遍。）

我们能发现，无论是上面的哪种代码，都与先前可逃句柄作用域中的样例代码有一个最大的区别：

- ① 这里代码的值在 C++ 层面并不是通过 `return` 返回的，而是在函数结束之前传入另一个函数的。待另一个函数执行完毕（如 `Set` 执行完毕或者 `Call` 执行完毕）之后，当前函数才真正结束，`HandleScope` 的生命周期才算结束。所以这些操作一直都是在 `HandleScope` 中进行的。而无论是 `Set` 还是 `Call`，它函数内部的实现必有办法把值获取到并且做一些事情，如复制对象、加引用等，使得原句柄被回收也无妨。
- ② 在可逃句柄作用域里面的样例代码中，生成出来的句柄要被返回到它的调用方。而调用方拿到该句柄的时候，就说明当前函数已经结束并退出，`HandleScope` 已经做了它的使命即回收句柄，所以需要有一个可逃句柄作用域来帮该句柄逃脱。

理解了生命周期之后，就不怕这类语义上的迷惑之处了，哪怕 `args.GetReturnValue().Set(...)` 的语义是告诉你它是设置 JavaScript 函数的返回值，它也不是把句柄返回 C++ 的上层函数去。就算是在 `args.GetReturnValue().Set(...)` 之前加上一个 `return` 也是一样的结果：

```
return args.GetReturnValue().Set(argv[0]);
```

因为这个 `argv[0]` 会被传到 `Set` 函数中，待 `Set` 执行完之后再返回该函数的返回结果 `void`。

最后，还有很重要的一点，在 Node.js 的事件循环中，这些代码全是以单线程的形式运行的，所以你不用考虑 `args.GetReturnValue().Set(...)` 中存在一些多线程操作这种情况。

解惑2

还是刚才的这段 `entry.cpp` 中的代码：实际上笔者把这个 `RunCallback` 函数中的 `HandleScope scope(isolate);` 去掉也没关系。

为什么这么说呢？实际上 Node.js 在调起 C++ 扩展之前就已经为我们在上层 C++ 函数中创建了一个句柄作用域了。

这里并不是说一个函数就需要有一个句柄作用域。句柄作用域会以栈的形式存在于 Chrome V8 实例中，并且栈顶的作用域是当前活动作用域。既然 Node.js 在调用之前就已经创建了一个句柄作用域，而且我们也确信这些函数的确只有 Node.js 会进行扩展调用，这也就坚信了它不会脱离 Node.js 那层作用域的掌控，那么我们为什么还要再套一层呢？

在图 3-2 中，图左的层级关系图就是原代码中句柄作用域的一个嵌套情况，cb 和 argv 数组句柄由 RunCallback 管控，当这个 C++ 函数结束的时候，这两个句柄都被回收，整个 Node.js 进行扩展调用的链路结束之后的结果就是这两个句柄被回收；图右的层级关系是函数中没有句柄作用域的情况，由于失去了句柄作用域的掌控，因此在 RunCallback 结束之后，这两个句柄并不会被回收，但是等到这个 Node.js 扩展调用链路结束之后，它会被 Node.js 层调用时的句柄作用域所回收。

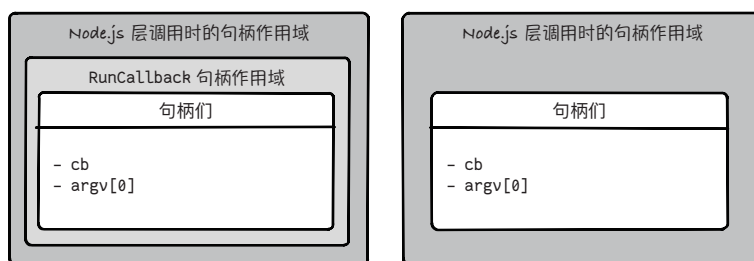


图 3-2 句柄作用域在 Node.js 函数中的对比

所以最终的效果是一样的，这些句柄只存在于这个调用链路中，不会跑到外面去。

可能很多人就会问了，那既然如此，肯定在整个程序中有一个最终的作用域。既然有最终作用域的话，我们为什么还要在自己的函数中声明句柄作用域呢？

这里又回到了垃圾回收这里。如果所有的句柄都在最终作用域中，我们几乎就可以认为这些句柄都不会被回收了。

所以我们需要按需求来声明句柄作用域，在必要的时候声明句柄作用域，使得一些句柄存活的生命周期是一个正常的值。

综上所述，通常在 C++ 扩展中：

- 能且只能直接被 Node.js 扩展链路调用的函数是不需要声明句柄作用域的，除非你需要创建非常多的对象，而这些对象的生命周期事实上又不需要存在于整个链路当中。
- 在通过 libuv 将不属于 Node.js 扩展链路调用的函数（如多线程的一些操作）硬插入事件循环的时候，由于通常它们并不在 Node.js 的句柄作用域掌控之中，因此通常情况下这些函数内部应该需要有句柄作用域来管理函数内部的句柄。

3.4.3 小结

句柄作用域就是管理这些句柄的一种类。它以栈的形式一层一层套着，存在于 Isolate 实例中，栈顶的作用域是当前活动作用域。每次新建对象时得到的句柄都会与当前活动作用域绑定。当活动作用域被析构时（通常是句柄作用域所处的 C++ 作用域结束导致生命周期到期所致），与其绑定的所有句柄都会被回收。这里有一种例外，就是可逃句柄作用域所设置的已逃脱句柄可以逃过一劫。

在 Node.js 进行 C++ 扩展调用时，会事先为其创建一个句柄作用域。所以开发者要按需决定是否还需要再自行创建句柄作用域来管理下级的一些句柄。

3.4.4 参考资料

[1] Embedder's Guide: <https://github.com/v8/v8/wiki/Embedder%27s-Guide>.

3.5 上下文（Context）

上下文是 Chrome V8 中的 JavaScript 代码执行环境。所以你想执行 JavaScript 代码的时候，必须为其指定一个上下文，就像 3.3 节中的示例代码一样。

```
Local<Script> script = Script::Compile(context, source).ToLocalChecked();
```

在 Chrome V8 中，除了 Isolate 实例是各自独立的，上下文也是独立且允许存在多个的。在同一个 Isolate 中，不同的上下文也是不相干的，其可以执行各自的 JavaScript 代码。

一个上下文为 JavaScript 的执行提供了内置的对象和方法，但多个上下文的意义何在呢？

读者来看看下面的代码吧：

```
Object.prototype.toString = function() {
  return " 蛋花汤汪汪汪 🐶, 南瓜饼喵喵喵 🐱";
};
```

上面的代码执行完后，上下文就被污染了，再去执行另一段与该代码完全无关的代码时，不就“挂”了吗？万一人家的代码是这样的：

```
Object.prototype.toString.call([]);
```

其本来预期返回的是一个 "[object Array]", 结果却返回一个 "蛋花汤汪汪汪 🐼, 南瓜饼喵喵喵 🐱", 那麻烦就大了。

这就说明了在一个程序中, 两段完全不相干的 JavaScript 代码期望跑在不同上下文中的必要性。这个时候又有读者要问了, 就算这样, 那我们在写 Node.js 的 C++ 扩展时, Node.js 总只有一个上下文了吧? 当然也不是啦。详见下面的介绍。

vm

Node.js 有一个非常有名的内置模块, 叫 vm。它的本质就是利用 Chrome V8 的可多上下文的特性。下面我们就来看看 vm 的实现吧。轻车熟路地打开 Node.js 6.9.4 版本的 Node.js 仓库¹ 中的 lib/vm.js 文件, 能找到这样的代码:

```
const binding = process.binding('contextify');
```

```
...
```

```
exports.createContext = function(sandbox) {
  if (sandbox === undefined) {
    sandbox = {};
  } else if (binding.isContext(sandbox)) {
    return sandbox;
  }

  binding.makeContext(sandbox);
  return sandbox;
};
```

其中 binding.makeContext 函数源自 src/node_contextify.cc 中的 MakeContext 函数。

```
static void MakeContext(const FunctionCallbackInfo<Value>& args) {
  Environment* env = Environment::GetCurrent(args);

  if (!args[0]->IsObject()) {
    return env->ThrowTypeError("sandbox argument must be an object.");
  }
  Local<Object> sandbox = args[0].As<Object>();

  // 不允许一个沙箱被多次上下文化
  CHECK(
    !sandbox->HasPrivate(
      env->context(),
      env->contextify_context_private_symbol()).FromJust());

  TryCatch try_catch(env->isolate());
  ContextifyContext* context = new ContextifyContext(env, sandbox);
```

¹ Node.js 的 v6.9.4 代码地址是 <https://github.com/nodejs/node/tree/v6.9.4>。

```

    if (try_catch.HasCaught()) {
        try_catch.ReThrow();
        return;
    }

    if (context->context().IsEmpty())
        return;

    sandbox->SetPrivate(
        env->context(),
        env->contextify_context_private_symbol(),
        External::New(env->isolate(), context));
}

```

从中可以看到重要的一点就是，这个函数创建了一个叫 ContextifyContext 的对象。我们继续驾轻就熟地去看看 ContextifyContext 的构造函数吧。

```

ContextifyContext(Environment* env, Local<Object> sandbox_obj) : env_(
    env) {
    Local<Context> v8_context = CreateV8Context(env, sandbox_obj);
    context_.Reset(env->isolate(), v8_context);

    // 分配失败，或者调用栈满了
    if (context_.IsEmpty())
        return;
    context_.SetWeak(this, WeakCallback, v8::WeakCallbackType::kParameter)
;
    context_.MarkIndependent();
}

```

我们能看到，在构造函数的最开始有一个上下文对象的本地句柄，它从 CreateV8Context 获取。现在似乎离答案很近了，我们还是继续钻进去看看吧。

```

Local<Context> CreateV8Context(Environment* env, Local<Object> sandbox_obj) {
    EscapableHandleScope scope(env->isolate());
    Local<FunctionTemplate> function_template =
        FunctionTemplate::New(env->isolate());
    function_template->SetHiddenPrototype(true);

    function_template->SetClassName(sandbox_obj->GetConstructorName());

    Local<ObjectTemplate> object_template =
        function_template->InstanceTemplate();

    NamedPropertyHandlerConfiguration config(...);
    object_template->SetHandler(config);
}

```

```

    Local<Context> ctx = Context::New(env->isolate(), nullptr, object_
template);

    if (ctx.IsEmpty()) {
        env->ThrowError("Could not instantiate context");
        return Local<Context>();
    }

    ctx->SetSecurityToken(env->context()->GetSecurityToken());

    // 我们需要把新建的上下文的生命周期与沙箱对象的生命周期绑在一起。原理是让它们各自
    // 引用对方
    // 1. 上下文能直接引用沙箱作为自己的一个内嵌字段
    // 2. 但是我们却无法直接在一个对象中直接引用上下文，所以这里“曲线救国”，引用
    // 上下文中的全局对象
    ctx->SetEmbedderData(kSandboxObjectIndex, sandbox_obj);
    sandbox_obj->SetPrivate(env->context(),
                           env->contextify_global_private_symbol(),
                           ctx->Global());

    env->AssignToContext(ctx);

    return scope.Escape(ctx);
}

```

怎么样？是不是看到了一些熟悉的内容？比如 `EscapleHandleScope` 等。当然这些不是本节要介绍的重点。本节的重点是，在这个函数中创建一个新的上下文，这个上下文将被用于 `vm` 执行沙箱代码。

值得一提的是，上面这段代码中有一个比较巧妙的写法——可以叫它“一根绳上的蚂蚱”。

沙箱对象句柄是一个本地句柄，由 `lib/vm.js` 的相应函数传入，它有一个自己的生命周期。而相应地，逃脱的上下文句柄也有一个生命周期。为了让它们的生命周期绑在一起，`Node.js` 硬生生地让它们互相引用了，引用的办法笔者已经写在了上面代码的注释中。

言归正传，从 CPU 运行时间与内存的角度来看，创建一个新的执行上下文的开销很大。但是不用担心，V8 的缓存机制让这个操作在第二次、第三次以及以后次数的时候，开销会变小很多。原因在于这个开销的大头是解析“创建内置对象的 JavaScript 代码”。在第一次上下文创建成功之后，下次创建就不需要再解析这些代码，而会直接执行这些代码来创建内置对象。

而且以 V8 的黑科技来说，如果再在编译 V8 的时候加入编译选项 `snapshot=yes` 的话，这些创建好的内置对象会被放入快照中，可在创建上下文的时候直接到快照中取。这就是

Chrome V8 高效的另一方面体现了——善用缓存。不过很显然，本书并不关心这些黑科技——这只是一本 Node.js 的 C++ 扩展开发手记而已。

3.6 模板（Template）

这里的模板可不是指 C++ 中的模板。Chrome V8 中的模板指的是在上下文中 JavaScript 对象以及函数的一个模具。你可以用一个模板来把 C++ 函数或者数据结构包裹进 JavaScript 的对象中，这样 JavaScript 代码就能对它做一些不可描述的事情了。

Node.js 的 C++ 核心模块就有被这样包裹的。Google Chrome 中的 DOM 节点实际上也是用 C++ 完成的，然后再用模板包裹成 JavaScript 对象，这样我们就能在浏览器中使用 JavaScript 来对它们进行操作了。

我们做 C++ 扩展实际上就是要让 C++ 代码能被 Node.js 中的 JavaScript 代码所调用。

本节介绍两种模板，这两种模板均继承自模板类（Template）：

- 函数模板（Function Template）；
- 对象模板（Object Template）。

3.6.1 函数模板（Function Template）

函数模板在 Chrome V8 中的数据类型是 `FunctionTemplate`。它是一个 JavaScript 函数的模具。

当生成一个函数模板后，我们通过调用它的 `GetFunction` 方法来获取其函数实体句柄，这个函数是能直接被 JavaScript 所调用的。

具体的用法大家可以阅读本书的随书代码“3. function template”中的 `template.cpp`。

```
// template.cpp 节选
void Method(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();
    args.GetReturnValue().Set(String::NewFromUtf8(isolate, "this is a
function"));
}

void init(Local<Object> exports)
{

```

```

Isolate* isolate = Isolate::GetCurrent();
HandleScope scope(isolate);
Local<FunctionTemplate> t = FunctionTemplate::New(isolate, Method);
Local<Function> fn = t->GetFunction();
Local<String> name = String::NewFromUtf8(isolate,
"funcCreateByTemplate");
    fn->SetName(name);
    exports->Set(name, fn);
}

NODE_MODULE(_template, init)

```

我们看到，Method 函数的返回值是 void 类型的，并且有一个单一的参数是 FunctionCallbackInfo<Value> 引用。细心的读者可能发现了，好像前两份用于在 JavaScript 中跑的 C++ 代码也是这样的一个声明结构。

没错，这种函数在 Node.js v6.9.4 以及其他的一些相应版本中叫 FunctionCallback 类型。在更早的版本中，用于跑在 JavaScript 中的 C++ 函数的结构就可能是其他样子了。这是因为 Chrome V8 的版本不同，导致底层所需要的一些数据类型也不一样——它经常做一些不兼容的升级，跟着升级的 Node.js 自然也就变了。

我们来看一看 FunctionCallback 类型的定义¹吧：

```
typedef void(* FunctionCallback) (const FunctionCallbackInfo< Value > &info)
```

也就是这样的一个函数：

```

void 函数名 (const FunctionCallbackInfo<Value>& 参数名)
{
}

```

继续往下看，根据代码 NODE_MODULE 宏得知，在 Node.js 加载这个 C++ 扩展模块的时候，会把 exports 对象传给 Init 函数执行。重点就在这个函数中。

前面两句实例和句柄作用域的代码就不再赘述了。

第三句代码 Local<FunctionTemplate> t = ... 就是指将 Method 这个函数进行包裹，产生一个函数模板，并返回它的句柄给 t。

FunctionTemplate::New 函数接收 5 个参数，其中 4 个都是有默认参数的。

¹ 关于 FunctionCallback 的定义，在 Node.js v6.9.4 所需要的 v5.1.281 版本的 Chrome V8 的 v8.h 中第 5119 行。

- ① Isolate: 函数模板所属的 Chrome V8 实例。
- ② FunctionCallback: 需要包裹的函数, 默认值为 0, 也就是 NULL。
- ③ Local<Value>: 默认为 Local<Value>()。
- ④ Local<Signature>: 默认为 Local<Signature>()。
- ⑤ int: 默认为 0。

通常情况下, 我们只关心前面两个参数。第一个参数是 Chrome V8 实例, 第二个就是我们要包裹的函数了。随书代码中传入的 FunctionCallback 就是事先准备好的 Method, 一个返回 "this is a function" 字符串的函数。

接下来我们通过 `t->GetFunction()` 来得到这个函数模板的一个 JavaScript 中能用的函数 (Function) 实例。最后笔者将这个函数实例命名为 `funcCreateByTemplate`, 并给 `exports` 这个对象也以 `funcCreateByTemplate` 的字段名挂上这个函数。

接下来我们就能在终端中编译这段代码, 然后运行 Node.js, 以便运行一下这个扩展了。

```
$ node-gyp rebuild
...
$ node
> const t = require("./build/Release/template");
undefined
> t.funcCreateByTemplate();
'this is a function'
```

我们通过执行 `require` 载入这个扩展进行初始化扩展, 并得到它的 `module.exports` 赋值给 `t`。这个时候 `t` 就被挂上了一个叫 `funcCreateByTemplate` 的函数, 它是从一个叫 Method 的 FunctionCallback 包裹并抽取出来的。

细心的读者可能又要问了, 为什么这一次代码的 `Init` 函数跟第一份和第二份的随书代码很不一样?

它们其实是一样的, 只不过在这一份代码中, 为了给大家展示 `FunctionTemplate`, 笔者将前面两份代码中的 `NODE_SET_METHOD` 宏和内联函数展开了。为了鼓励大家追根溯源的态度, 笔者在这里把这个宏的定义向大家展示一下¹。

```
inline void node::NODE_SET_METHOD(v8::Local<v8::Object> recv,
                                   const char* name,
                                   v8::FunctionCallback callback) {
```

¹ 这段宏的展开在 Node.js v6.9.4 代码的 `src/node.h` 中可以找到。

```
v8::Isolate* isolate = v8::Isolate::GetCurrent();
v8::HandleScope handle_scope(isolate);
v8::Local<v8::FunctionTemplate> t = v8::FunctionTemplate::New(isolate,
                                                                callback);

v8::Local<v8::Function> fn = t->GetFunction();
v8::Local<v8::String> fn_name = v8::String::NewFromUtf8(isolate, name);
fn->SetName(fn_name);
recv->Set(fn_name, fn);
}

#define NODE_SET_METHOD node::NODE_SET_METHOD
```

所以说，如果读者以前写过 Node.js 的 C++ 扩展，或者尝试阅读、修改、执行过本书前面一些章节相关的随书代码，你其实就已经接触过 FunctionTemplate 了。

接下来笔者介绍几个常用的 FunctionTemplate API。

1. 创建 (New)

通常情况下，我们通过 FunctionTemplate::New 这个类的静态函数来生成一个函数模板，并且返回这个模板的本地句柄。

实际上 New 有 5 个参数，但是后 4 个参数都有默认值，且通常我们在声明的时候就只用到前 3 个参数，如表 3-1 所示。由于篇幅限制，本书仅介绍前 3 个参数。

表 3-1 static FunctionTemplate::New 的常用参数表

参数位置	参数名	参数类型	默认值	说明
0	isolate	Isolate*		Isolate 实例
1	callback	FunctionCallback	0	本书经常提到的 void 函数名 (const FunctionCallbackInfo<Value>& args) 函数
2	data	Local<Value>	Local<Value>	该参数会被传入你写的 FunctionCallback 中，可通过 args.Data() 在函数中获取

callback 即函数本体，本书已经介绍过很多次了，就是如下这样的函数：

```
void 函数名 (const FunctionCallbackInfo<Value>& args)
{
}
}
```


当函数模板被实例化成函数时,一旦调用这个函数,data 就会随调用一起传入函数本体中。开发者可以在自己写的函数本体中通过 `args.Data()` 获取到这个预传的数据。

2. 设置本体 (SetCallHandler)

这个函数用于设置函数模板的函数本体。

因为在一个函数模板中,在调用 New 创建时,callback 也是可以不传的,这个时候生成的函数模板其实是不带函数本体的。

执行 SetHandler 就可以后知后觉地为我们设置函数本体,也可以替换原有的函数本体。这个函数通常用到两个参数,可以对应到 New 中,如表 3-2 所示。

表 3-2 FunctionTemplate::SetCallHandler 的常用参数表

参数位置	参数名	参数类型	默认值
0	callback	FunctionCallback	
1	data	Local<Value>	Local<Value>()

3. 设置类名 (SetClassName) 与设置参数数量 (SetLength)

这两个函数,前者是当这个函数模板实例化出来的函数若作为一个构造函数的话,用它通过 new 操作符实例化出来的对象对应的类名就是通过 SetClassName 设定的。

```
// `SetClassName` 样例
Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate);
tpl->SetClassName(String::NewFromUtf8("ClassName"));
```

SetLength 则是设置这个函数在定义时的参数数量。这就相当于:

```
function test(a, b) {
  // ..
}

console.log(test.length);
```

这段代码的 test 函数输出的长度就是 2,因为其在定义时有两个参数。

```
// `SetLength` 样例
Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate);
tpl->SetLength(2);
```

4. 实例化 (GetFunction)

无论函数模板怎么定义，不出意外的话，最终它们都是要为实例化成一个 JavaScript 中可用的函数而服务的。所以通常一个函数模板经过各种设置之后，都要实例化成一个函数以供使用。

实例化的过程很简单，只要调用一个函数模板的 `GetFunction` 函数就能得到了。

```
Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate);
MaybeLocal<Function> func = tpl->GetFunction();
```

值得注意的是，这个 `GetFunction()` 函数返回的是一个待实本地句柄，所以最后我们还要通过 `ToLocalChecked()` 将其本地化。

其实 `GetFunction()` 也有一个重载是直接返回本地句柄。这是旧版本时的 API 了。在新版 Chrome V8 的 API 中，这个返回本地句柄的重载已经被标记为“快要被弃用 (V8_DEPRECATED_SOON) 的 API”了。所以还是推荐先获取待实本地句柄，然后再进行本地化。

3.6.2 对象模板 (Object Template)

在 Chrome V8 的官方文档中，对于对象模板的阐述是这样的：对象模板用于在运行时创建对象。从对象模板被创建的对象会被挂上被加到这个模板中的属性。

对象模板类名是 `ObjectTemplate`。`ObjectTemplate` 和 `FunctionTemplate` 均继承自 `Template`。

对象模板有几种常见的用法。

- ① 每个函数模板都会有一个与之相关联的对象模板。当该函数模板被用作一个构造函数时，这个对象模板就用来配置创建出来的对象。而这个对象模板有一个大家都比较熟悉的名字——原型链模板 (`PrototypeTemplate`)，原型链模板实际上就是一个对象模板，只不过是在函数模板中作为其原型链用的。

我们来看看随书代码“4. object template”中 `template.cpp` 的一个片段：

```
void Constructor(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();
    args.This()->Set(String::NewFromUtf8(isolate, "value"),
        Number::New(isolate, 233));
    return args.GetReturnValue().Set(args.This());
}
```

```

void ClassGet(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();
    return args.GetReturnValue().Set(args.This()->Get(String::NewFromUtf8(
isolate, "value")));
}

void Init(Local<Object> exports)
{
    Isolate* isolate = Isolate::GetCurrent();
    HandleScope scope(isolate);

    Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate,
Constructor);
    tpl->SetClassName(String::NewFromUtf8(isolate, "TestClass"));
    Local<ObjectTemplate> proto = tpl->PrototypeTemplate();
    proto->Set(String::NewFromUtf8(isolate, "get"),
FunctionTemplate::New(isolate, ClassGet));

    exports->Set(String::NewFromUtf8(isolate, "TestClass"), tpl-
>GetFunction());

    // ...
}

```

首先在载入 Init 阶段我们从 Constructor 声明了一个函数模板，让它作为一个构造函数使用。这个构造函数如果改用 JavaScript 编写，基本上是这样的：

```

function TestClass() {
    this.value = 233;
}

```

接下来我们拿到它的原型链模板，就是笔者先前说的对象模板了。我们在它上面挂上一个叫 get 的函数，这个函数内部实现是返回 this 的 value 值。如果用 JavaScript 编写，大概是这样的：

```

const proto = TestClass.prototype;
proto.get = function() {
    return this.value;
};

```

最后，通过 tpl->GetFunction() 把这个函数模板实例化成一个真真正正存在的函数，而且是一个构造函数。将其挂载到 exports 对象下的 TestClass 字段中。

编译执行这个目录下的代码，大家能体验到这样一个结果：

```
$ node-gyp rebuild
...
$ node
> const t = require("../build/Release/template");
undefined
> t.TestClass
[Function: TestClass]
> var a = new t.TestClass()
undefined
> a
TestClass { value: 233 }
> a.get()
233
> a.value = 1
1
> a.get()
1
```

这就是对象模板作为一个类的原型链时的用处。

② 对象模板也能脱离函数模板使用。跟函数模板差不多，用这个对象模板的模子创建一个对象。我们继续看“4. object template”中 `template.cpp` 的代码片段。

```
Local<FunctionTemplate> fun = FunctionTemplate::New(isolate);
fun->SetClassName(String::NewFromUtf8(isolate, "TestClass2"));
Local<ObjectTemplate> obj_tpl = ObjectTemplate::New(isolate, fun);
obj_tpl->Set(String::NewFromUtf8(isolate, "num"), Number::New(isolate,
233));
Local<Array> array = Array::New(isolate, 10);
for(int i = 0; i < 10; i++)
{
    array->Set(Number::New(isolate, i), obj_tpl->NewInstance());
}
exports->Set(String::NewFromUtf8(isolate, "array"), array);
```

笔者创建了一个对象模板 `obj_tpl`，并且传入 `fun` 用于以后显示这个对象所属的类名。实际上 `ObjectTemplate::New` 函数有两个参数。

- Isolate 实例。
- 可选参数，这是一个函数模板。默认不传，这种情况就代表一个空的函数模板 `Local<FunctionTemplate>()`。

在创建完这个对象模板之后，笔者往里面加一个 `num`，其值为 233。接下来创建了一个长度为 10 的数组，往里面依次塞入对象，每个对象都是通过 `obj_tpl->NewInstance()` 实例化过来的。

要注意的是，新版 Chrome V8 推荐 `obj_tpl->NewInstance()` 的返回结果是待实例本地句柄，所以在得到结果后再通过 `ToLocalChecked()` 进行本地化。

```
MaybeLocal<Object> maybe_local_handler = obj_tpl->NewInstance();
Local<Object> local_handler = maybe_local_handler.ToLocalChecked();
```

最后把这个数组挂载到 `exports` 下的 `array` 字段中。

编译执行这个目录下的代码，大家能体验到这样的一个结果：

```
$ node-gyp rebuild
...
$ node
> const t = require("../build/Release/template");
undefined
> t.array
[ TestClass2 { num: 233 },
  TestClass2 { num: 233 },
  TestClass2 { num: 233 },
  TestClass2 { num: 233 },
  TestClass2 { num: 233 },
  TestClass2 { num: 233 },
  TestClass2 { num: 233 },
  TestClass2 { num: 233 },
  TestClass2 { num: 233 },
  TestClass2 { num: 233 } ]
> t.array[0].num = 2333
2333
> t.array
[ TestClass2 { num: 2333 },
  TestClass2 { num: 233 },
  TestClass2 { num: 233 },
  TestClass2 { num: 233 },
  TestClass2 { num: 233 },
  TestClass2 { num: 233 },
  TestClass2 { num: 233 },
  TestClass2 { num: 233 },
  TestClass2 { num: 233 },
  TestClass2 { num: 233 } ]
```

可以看到，数组中是一个对象，并且在刚创建的时候都是 `TestClass2 { num: 233 }`。

当我们修改任意一个 `TestClass2` 对象中的 `num` 值时，发现其他对象并未被更改——这说明了它们本来就是不同的实体。对象模板根据自身的模子，通过 `NewInstance` 函数来实例化出外观一致的不同实体。

这种用法也通常出现在创建上下文的时候。传入的 `global` 对象通常也都是由对象模板实例化出来的。比如前面提及的 Node.js 代码中 `src/node_contextify.cc` 就有这样的代码，上下文创建时传入一个模板对象作为其第 3 个参数。

```
Local<Context> ctx = Context::New(env->isolate(), nullptr, object_
template);
```

读者也能在参考资料 *Embedder's Guide* 中找到对象模板用作上下文的全局对象的例子。

```
Local<ObjectTemplate> global = ObjectTemplate::New(isolate);
global->Set(String::NewFromUtf8(isolate, "log"),
FunctionTemplate::New(isolate, LogCallback));

Persistent<Context> context = Context::New(isolate, NULL, global);
```

接下来笔者介绍 `ObjectTemplate` 的几个常用 API。

1. 创建 (New)

这个 API 在前面就已经介绍了，其有两个参数：一个是 `Isolate` 实例，另一个就是能与之绑定的构造函数。

2. 设置函数体 (SetCallAsFunctionHandler)

有些读者可能会有疑问了，这明明是一个对象的模板，为什么又能有函数体呢？其实在 JavaScript 当中，函数和对象的定界是比较模糊的。

我们来看下面这段代码：

```
function func() {
    return 2333;
}

func.cat = "南瓜饼";
func.dog = "蛋花汤";

console.log(func);
console.log(func());
console.log(func.cat);
console.log(func.dog);
```

它输出的结果是：

```
{ [Function: func] cat: '南瓜饼', dog: '蛋花汤' }
2333
南瓜饼
蛋花汤
```

怎么样？是不是调用的时候是一个函数，但是平时也能当作对象来用它的一些属性？实际上一个 JavaScript 中还有很多隐藏的属性，比如 `func.length` 代表这个函数在定义时的参数个数。

所以设置函数体就是将这个对象模板配置一下，让它实例化出来的对象与上述行为相似。

`SetCallAsFunctionHandler` 接收两个参数，与 `FunctionTemplate::SetCallHandler` 参数一样，分别是函数本体与将传入函数本体的 `data`。

接下来我们就继续对“4. object template”中的 `template.cpp` 代码片段进行解析吧，看看笔者是如何用 C++ 的代码来实现上面那样的效果的。

笔者事先准备好了一个函数，与前面所述的返回值为 2333 的函数相对应。

```
void Func(const FunctionCallbackInfo<Value>& args)
{
    args.GetReturnValue().Set(2333);
}
```

然后在载入模块时的 `Init` 函数中生成一个包含 "cat" 和 "dog" 的对象模板，再将这个 `Func` 函数通过 `SetCallAsFunctionHandler` 挂载到对象模板中，最后实例化对象模板。

```
void Init(Local<Object> exports)
{
    ...

    // 设置函数体
    Local<ObjectTemplate> obj_with_func_tpl = ObjectTemplate::New(isolate);
    obj_with_func_tpl->Set(String::NewFromUtf8(isolate, "cat"),
        String::NewFromUtf8(isolate, "南瓜饼"));
    obj_with_func_tpl->Set(String::NewFromUtf8(isolate, "dog"),
        String::NewFromUtf8(isolate, "蛋花汤"));
    obj_with_func_tpl->SetCallAsFunctionHandler(Func);
    exports->Set(String::NewFromUtf8(isolate, "func"), obj_with_func_tpl-
>NewInstance());
}
```

好的，下面跑一遍代码吧。

```
$ node-gyp rebuild
...
$ node
> const a = require("./build/Release/template");
undefined
> a.func
{ [Function] cat: '南瓜饼 ', dog: '蛋花汤' }
> a.func()
2333
```

怎么样，是不是跟刚才那段 JavaScript 代码的效果一样？这就是 SetCallAsFunction-Handler 的用法了。

3.6.3 对象模板的访问器（Accessor）与拦截器（Interceptor）

访问器与拦截器是对象模板中两种不同的 C++ 回调函数：

- 访问器的回调函数会在模板对象生成的对象中指定属性被访问时执行；
- 拦截器的回调函数会在模板对象生成的对象中任何属性被访问时执行。

1. 访问器（Accessor）

访问器其实不是模板中的专属品，但是放在本节中介绍也是可以的。

还记得 JavaScript 中的 getter 和 setter 吗，这就是访问器了。在 JavaScript 中设置一个访问器的代码是这样的：

```
const pet = {
  name: "蛋花汤",
  type: "🐶",
  get kind() {
    return this.type === "🐶" ? "狗" : "外星人";
  }

  set kind(type) {
    return this.type = type === "狗" ? "🐶" : "😱";
  }
};

pet.kind = "变异";
console.log(pet.type); // 🐶
```


而在使用 Chrome V8 的 C++ 代码中，一样能为对象模板或者对象创建一个访问器，使用 `ObjectTemplate` 的 `SetAccessor` 函数。其参数如表 3-3 所示。

表 3-3 `ObjectTemplate::SetAccessor` 参数表

顺序	参数	类型	说明	默认值
0	name	<code>Local<String></code>	访问器的属性名	
1	getter	<code>AccessorGetterCallback</code>	访问器的 <code>get</code> 函数	
2	setter	<code>AccessorSetterCallback</code>	访问器的 <code>set</code> 函数	0 或者 <code>NULL</code>
3	data	<code>Local<value></code>	当访问器被调用的时候，这个 <code>data</code> 就会被传进去	<code>Local<Value>()</code>
4	settings	<code>AccessControl</code>	访问器的权限控制，是否允许跨上下文访问。这里有 <code>DEFAULT = 0</code> 、 <code>ALL_CAN_READ = 1</code> 和 <code>ALL_CAN_WRITE = 2</code> 以及 <code>ALL_CAN_READ ALL_CAN_WRITE = 3</code> 四个值	0
5	attribute	<code>PropertyAttribute</code>	访问器属性，枚举类型，这里有 <code>None</code> 、 <code>ReadOnly</code> 、 <code>DontEnum</code> 和 <code>DontDelete</code> 四个值	<code>None</code>
6	signature	<code>Local<AccessorSignature></code>	签名器，用于描述访问器的有效接收者，本书不做赘述	<code>Local<AccessorSignature>()</code>

在我们的平日开发中，通常只关心前 3 个参数——`name`、`getter` 和 `setter`。其他参数在本书中就不做赘述了。

在下一步介绍之前，这里再给出 `AccessorGetterCallback` 和 `AccessorSetterCallback` 的定义。

```
typedef void(* AccessorGetterCallback)(Local<String> property, const
PropertyCallbackInfo<Value> &info);
typedef void(* AccessorSetterCallback)(Local<String> property, Local<Value>
value, const PropertyCallbackInfo<void> &info)
```

这里需要实现的两个函数都是以属性名作为第一个参数。在 `setter` 中的第二个参数是当前要设置的值，它与 `getter` 的最后一个参数都是一个 `PropertyCallbackInfo<Value>`，表示当前访问器调用时的一些必要信息，其中重要的内容有 `info.This()`、`info.Data()` 和 `info.Holder()`。`setter` 的最后一个参数是 `PropertyCallbackInfo<void>`。

`PropertyCallbackInfo<T>` 的几个重要属性如下。

- `info.This()`：代表 JavaScript 中的 `this`。

- `info.Data()`: 代表设置访问器时的第 4 个参数。
- `info.Holder()`: 这个访问器的真实拥有者, 在 `this` 没有重新指定的情况下, 可以理解为它与 `info.This()` 相同。

我们使用访问器的时候, 通常根据需要来进行不同的操作。

比如, 如果我们写了一个静态的对象 (非类的实例), 那么它的访问器基本上也是全局静态的; 但如果一个访问器是在一个类的实例中, 那么对它的访问就需要每个实例都有自己的结果了。

(1) 访问静态全局变量

请打开随书代码 “5. object template accessor” 的 `accessor.cpp`, 跟着笔者来阅读一些片段。

```
int x = 0;

void Getter1(Local<String> property, const PropertyCallbackInfo<Value>&
info)
{
    info.GetReturnValue().Set(x);
}

void Setter1(Local<String> property, Local<Value> value, const
PropertyCallbackInfo<void>& info)
{
    x = value->Int32Value();
}

void Init(Local<Object> exports, Local<Object> module)
{
    Isolate* isolate = Isolate::GetCurrent();
    HandleScope scope(isolate);

    Local<ObjectTemplate> tpl = ObjectTemplate::New(isolate);
    tpl->SetAccessor(String::NewFromUtf8(isolate, "var1"), Getter1,
Setter1);

    ...

    Local<Object> ret = tpl->NewInstance();
    ...

    module->Set(String::NewFromUtf8(isolate, "exports"), tpl-
>NewInstance());
}
```

编译执行上面的代码，读者能有这样的体验：

```
$ node-gyp
...
$ node
> const a = require("./build/Release/accessor");
undefined
> a.var1
0
> a.var1 = 233;
233
> a.var1
233
```

笔者首先写了两个访问器的函数，然后将 getter 和 setter 赋给对象模板的 var1 字段。最后将对象模板实例化成一个对象挂到 module.exports 下。

访问器的逻辑很简单，就是 C++ 代码中有一个文件中的全局变量 x，初始值为 0。在访问器的 getter 中，我们将 x 的值传给 info.GetReturnValue()，使其返回值设置为 x 的值；在访问器的 setter 中，我们则是把参数传进来的值 value 通过 value->Int32Value() 函数将这个 JavaScript 的数据对象转换为 C++ 底层的 int 类型，并且赋值给 x。这就是用访问器访问静态的全局变量。

还有一种类似的写法，就是将这个需要的值具象化成一个 JavaScript 的数据，然后在创建访问器的时候将这个数据放到访问器创建函数的第 4 个参数中。

下面是刚才那个源码文件的一个片段：

```
void Getter2(Local<String> property, const PropertyCallbackInfo<Value>&
info)
{
    info.GetReturnValue().Set(info.Data()->ToObject()-
>Get(String::NewFromUtf8(info.GetIsolate(), "inner")));
}

void Setter2(Local<String> property, Local<Value> value, const
PropertyCallbackInfo<void>& info)
{
    info.Data()->ToObject()->Set(String::NewFromUtf8(info.GetIsolate(),
"inner"), value);
}

void Init(Local<Object> exports, Local<Object> module)
{
```

```

Isolate* isolate = Isolate::GetCurrent();
HandleScope scope(isolate);

Local<ObjectTemplate> tpl = ObjectTemplate::New(isolate);
...

Local<Object> inner = Object::New(isolate);
inner->Set(String::NewFromUtf8(isolate, "inner"),
String::NewFromUtf8(isolate, "蛋花汤"));
tpl->SetAccessor(String::NewFromUtf8(isolate, "var2"), Getter2, Setter2,
inner);

Local<Object> ret = tpl->NewInstance();
...

module->Set(String::NewFromUtf8(isolate, "exports"), ret);
}

```

笔者创建了一个 `inner` 对象，它的初始值是 `{ inner: "蛋花汤" }`。将这个对象传入创建访问器的函数，那么以后每次进入访问器函数的时候，这个 `inner` 对象的句柄都可以使用 `info.Data()` 获取到。

通过命令行编译执行这个代码片段，读者能得到这样的体验：

```

$ node-gyp rebuild
...
$ node
> const a = require("../build/Release/accessor")
undefined
> a.var2
'蛋花汤'
> a.var2 = "南瓜饼";
'南瓜饼'
> a.var2
'南瓜饼'

```

不过 `info.Data()` 返回的是一个 `Local<Value>`，读者需要通过 `info.Data().ToObject()` 来获取它的对象本地句柄形态。

这种写法跟上面的写法类似，都是将状态存于一个指定的地方，然后固定地去取它。

(2) 访问动态变量

访问动态变量就是另一种写法了，我们能以一种隐匿的形式将这种动态变量藏于对象中，在 JavaScript 中使用的时候就好像它是一个私有变量一样，只能通过访问器访问。

为了方便讲解，本节用了 3.7.5 节会讲到的一个函数，就是 Object 的 SetInternalField。这个函数的大意就是往 Object 中塞一些内置的内存数据，该数据只能通过调用 Object 的 GetInternalField 函数获得，所以在 JavaScript 代码中无法获取这些内容——就算获取到了也无法识别，因为它们可以是 C++ 中的任意数据结构。

话不多说，先看“5. object template accessor”中 accessor.cpp 的代码片段：

```
class TestExternal {
public:
    TestExternal(Local<Object> obj)
    {
        value = 233;
        _handle.Reset(Isolate::GetCurrent(), obj);
        _handle.SetWeak(this, WeakCallback, v8::WeakCallbackType::kParameter
    );
        _handle.MarkIndependent();
    }

    ~TestExternal()
    {
    }

    void Set(int _value)
    {
        value = _value;
    }

    int Get()
    {
        return value;
    }

    Persistent<Object> _handle;

    static void WeakCallback(const v8::WeakCallbackInfo<TestExternal>& data)
    {
        TestExternal* ex = data.GetParameter();
        ex->_handle.Reset();
        delete ex;
    }

private:
    int value;
};
```

```

void Getter3(Local<String> property, const PropertyCallbackInfo<Value>&
info)
{
    Local<Object> self = info.Holder();
    Local<External> wrap = Local<External>::Cast(self->GetInternalField(0));
    void* ptr = wrap->Value();
    TestExternal* ex = (TestExternal*)ptr;

    info.GetReturnValue().Set(ex->Get());
}

void Setter3(Local<String> property, Local<Value> value, const
PropertyCallbackInfo<void>& info)
{
    Local<Object> self = info.Holder();
    Local<External> wrap = Local<External>::Cast(self->GetInternalField(0));
    void* ptr = wrap->Value();
    TestExternal* ex = (TestExternal*)ptr;

    ex->Set(value->ToInt32()->Value());
}

void Init(Local<Object> exports, Local<Object> module)
{
    ...

    // 动态变量
    tpl->SetAccessor(String::NewFromUtf8(isolate, "var3"), Getter3,
Setter3);

    Local<Object> ret = tpl->NewInstance();
    TestExternal* ex = new TestExternal(ret);
    ret->SetInternalField(0, External::New(isolate, ex));

    ...
}

```

在本次样例中，笔者为通过对象模板实例化出来的对象塞入了一个 `ExternalTest` 类的内置内存指针。

```

// 塞入内置内存块
ret->SetInternalField(0, External::New(isolate, ex));

```

在 var3 这个访问器的设置与获取的回调中拿出这个内存指针转化成 ExternalTest，然后拿它的 value 数据做文章。

```
void Getter3(Local<String> property, const PropertyCallbackInfo<Value>&
info)
{
    // 获取 `ExternalTest` 对象指针，并将它的 `Get()` 值作为返回结果
    Local<Object> self = info.Holder();
    Local<External> wrap = Local<External>::Cast(self->GetInternalField(0));
    void* ptr = wrap->Value();
    TestExternal* ex = (TestExternal*)ptr;

    info.GetReturnValue().Set(ex->Get());
}

void Setter3(Local<String> property, Local<Value> value, const
PropertyCallbackInfo<void>& info)
{
    // 获取 `ExternalTest` 对象指针，并将需要设置的值通过 `Set()` 设置到
    // `ExternalTest` 对象中
    Local<Object> self = info.Holder();
    Local<External> wrap = Local<External>::Cast(self->GetInternalField(0));
    void* ptr = wrap->Value();
    TestExternal* ex = (TestExternal*)ptr;

    ex->Set(value->ToUint32()->Value());
}
```

同时，为了管理 TestExternal 对象 ex 的生命周期，笔者还做了如下处理以防止内存泄漏：

- ① 在 ExternalTest 的构造函数中，把将要被挂载到 exports 下的 Local<Object> 对象句柄升格成一个持久句柄。

```
_handle.Reset(Isolate::GetCurrent(), obj);
_handle.SetWeak(this, WeakCallback, v8::WeakCallbackType::kParameter);
_handle.MarkIndependent();
```

- ② 在升格的时候传入回调函数 WeakCallback，以保证这个句柄在挂掉的时候去调用它。
- ③ 在 WeakCallback 被调用的时候，也就是这个句柄挂掉的时候，在该函数内部将 ExternalTest 的这个对象 ex 删除。

关于 InternalField 更具体的说明会在 3.7.5 节中给出。

最后，我们编译执行代码。

```
$ node-gyp rebuild
...
$ node
> const a = require("./build/Release/accessor")
undefined
> a.var3
233
> a.var3 = 2333;
2333
> a.var3
2333
```

2. 拦截器（Interceptor）

拦截器与访问器有相似之处，所以才把它们归到同一节目录下。只不过访问器是针对某个特定的访问设置的 getter 和 setter，而拦截器则是对于一个对象实例的所有相关访问进行拦截。

目前在 Chrome V8 中，一个对象模板有两种不同类型的拦截器可以设置。

- **映射型拦截器（Named Property Interceptor）**：当对于一个对象内成员的访问方式是字符串型的属性名时，映射型拦截器就会生效。举一个例子，在 Chrome 浏览器中，文档中的一些访问就是映射型拦截器，如 `document.theFormName.elementName`。
- **索引型拦截器（Indexed Property Interceptor）**：与映射型拦截器不同，索引型拦截器的访问与数组类似，通过整型下标来对内容进行访问。例如，在 Chrome 浏览器中，`document.forms.elements[0]` 这种形式的访问就是索引型拦截器的一种体现。

对象模板通过 `SetHandler` 来对这个模板设置拦截器。通过传入不同类型的配置对象来决定设置的是映射型拦截器还是索引型拦截器。其实还可以通过 `SetNamedPropertyHandler` 和 `SetIndexedPropertyHandler` 函数进行设置，不过这两个函数已经在现在的 Chrome V8 版本中被标记为待弃用了，不推荐使用。

下面来看看 `SetHandler` 函数的两个原型：

```
void ObjectTemplate::SetHandler(const NamedPropertyHandlerConfiguration&
configuration);
void ObjectTemplate::SetHandler(const IndexedPropertyHandlerConfiguration&
configuration);
```


(1) 映射型拦截器 (Named Property Interceptor)

Node.js 中的 `process.env` 就是通过模板对象加上映射型拦截器实例化出来的对象。

在观看 Node.js 源码时，笔者解释一下 `NamedPropertyHandlerConfiguration`。这个类的对象用于配置一个映射型拦截器。

我们不关心它内部是怎样的，但是需要知道它的构造函数。

```
NamedPropertyHandlerConfiguration::NamedPropertyHandlerConfiguration(
    GenericNamedPropertyGetterCallback getter = 0,
    GenericNamedPropertySetterCallback setter = 0,
    GenericNamedPropertyQueryCallback query = 0,
    GenericNamedPropertyDeleterCallback deleter = 0,
    GenericNamedPropertyEnumeratorCallback enumerator = 0,
    Local<Value> data = Local<Value>(),
    PropertyHandlerFlags flags = PropertyHandlerFlags::kNone);
```

- `getter`: 这是拦截器的 `getter` 函数，这个函数是这样的——`void 函数名 (Local<Name> property, const PropertyCallbackInfo<Value>& info)`。其在函数内部为 `info` 返回 `getter` 的值。
- `setter`: 这是拦截器的 `setter` 函数，这个函数是这样的——`void 函数名 (Local<Name> property, Local<Value> value, const PropertyCallbackInfo<Value>& info)`。其在函数内部把 `value` 的值设置到相应的地方。
- `query`: 用于对象内查询某属性状态的函数，如只读、不可枚举等，这个函数是这样的——`void 函数名 (Local<Name> property, const PropertyCallbackInfo<Integer>& info)`。其在函数内部为 `info` 返回一个 `Local<Number>` 的值，代表它的状态，如 `v8::ReadOnly`、`v8::DontDelete`、`v8::None` 等。
- `deleter`: 用于对象内删除属性的函数，这个函数是这样的——`void 函数名 (Local<Name> property, const PropertyCallbackInfo<Boolean>& info)`。其在函数内部做相应的删除操作之后为 `info` 返回一个是否可删除的 `Local<Boolean>` 布尔值。
- `enumerator`: 用于对象枚举，如定义了 `for...in`、`console.log` 等执行结果的行为等，这个函数是这样的——`void 函数名 (const PropertyCallbackInfo<Array>& info)`。其在函数内部为 `info` 返回一个字段数组，表示这个对象可枚举出来的字段名。
- `data`: 这个参数将会被传入上述的各种函数中供开发者使用。在上述各函数的 `PropertyCallbackInfo` 参数的对象中，有一个 `Data()` 函数就是用来获取这个 `data` 用的，如 `info.Data()`。

- flags: 表示这个拦截器的一些标识。PropertyHandlerFlags 这个枚举类型有 4 个值。
- kNone: 无标识。
- kAllCanRead: 所有属性可读。
- kNonMasking: 无遮罩模式, 对象上面的常规属性拥有更高的优先级。
- kOnlyInterceptStrings: 只拦截字符串。

下面先来看看 Node.js 6.9.4 版本中 src/node.cc 文件的片段。

```
void SetupProcessObject(...) {  
    ...  
  
    Local<ObjectTemplate> process_env_template =  
        ObjectTemplate::New(env->isolate());  
    process_env_template->SetHandler(NamedPropertyHandlerConfiguration(  
        EnvGetter,  
        EnvSetter,  
        EnvQuery,  
        EnvDeleter,  
        EnvEnumerator,  
        env->as_external(),  
        PropertyHandlerFlags::kOnlyInterceptStrings));  
  
    Local<Object> process_env =  
        process_env_template->NewInstance(env->context()).ToLocalChecked();  
    process->Set(env->env_string(), process_env);  
  
    ...  
}
```

这段代码的目的明显, 就是创建一个 process.env 对象模板, 然后给它设置映射型拦截器, 最后实例化出 env 对象, 然后将其挂到 process 对象上。

在这次调用中, 所有参数都没有用默认参数, 而是实打实地传了一堆有用的参数进去。其中回调函数就有 EnvGetter、EnvSetter、EnvQuery、EnvDeleter 和 EnvEnumerator。data 的值是 env->as_external(), 拦截标识为只拦截字符串。

接下来一个一个解析这个 env 拦截器中的各种回调函数。为了简化代码片段, env 拦截器的各回调函数中只保留 UNIX 部分的代码。

• GenericNamedPropertyGetterCallback

在 `process.env` 拦截器中，获取属性的回调函数是 `EnvGetter`。下面先来观察一下 `EnvGetter` 函数的本体。

```
static void EnvGetter(Local<Name> property,
                    const PropertyCallbackInfo<Value>& info) {
  Isolate* isolate = info.GetIsolate();

  node::Utf8Value key(isolate, property);
  const char* val = getenv(*key);
  if (val) {
    return info.GetReturnValue().Set(String::NewFromUtf8(isolate, val));
  }
}
```

下面逐行解读：

- ① 获取 `Isolate` 对象。
- ② 通过 `property` 生成一个 Node.js 中的 UTF8 字符串赋给 `key`。
- ③ 调用 `key` 的 `*` 操作符，得到一个字符串，并将该字符串传给 UNIX 中的系统 API——`getenv()`，从而得到环境变量的字符串。
- ④ 进入条件分支：
 - (a) 如果获取环境变量成功，就将这个环境变量转换为 `v8::String` 类型，并设置给 `info.GetReturnValue()`，表示返回这个环境变量的字符串。
 - (b) 如果获取环境变量失败，就不做任何事，实际上在 JavaScript 代码中的结果就是没有返回值，即 `undefined`。

这个结果是我们在 JavaScript 中访问 `process.env` 中任意一个有效对象会调用的函数。举一个例子，假设环境变量中有 `USER` 这个变量，那么 `process.env.USER` 其实就是进入了这个拦截器，且传入的 `property` 的最终值是 `"USER"`，而且这个函数的返回值就是环境变量中 `USER` 的相应值。

• GenericNamedPropertySetterCallback

接下来我们观察一下 `EnvSetter` 这个设置函数的本体。

```
static void EnvSetter(Local<Name> property,
                    Local<Value> value,
                    const PropertyCallbackInfo<Value>& info) {
```

```
node::Utf8Value key(info.GetIsolate(), property);
node::Utf8Value val(info.GetIsolate(), value);
setenv(*key, *val, 1);

// 无论是否成功, 总是返回 value
info.GetReturnValue().Set(value);
}
```

继续逐步解析:

- ① 将 `property` 转换为 Node.js 中的 UTF8 字符串, `value` 也做同样的操作。
- ② 调用系统 API `setenv`, 将键名和键值的 `char*` 字符串传入。
- ③ 将传入的 `value` 原封不动地返回。

我们在做 `process.env.USER = "foo"` 的时候就会调用这个函数, 正如我们所想, 它最终会调用 `setenv("USER", "foo", 1)`。

• GenericNamedPropertyQueryCallback

笔者前面讲过, 这个函数返回某个属性的状态。

熟悉 JavaScript API 的读者都知道, 当我们为一个对象通过 `defineProperty` 新增属性的时候, 其传入的第 3 个参数中有 `enumerable`、`writable` 等属性, 表示这个新增的属性是否可枚举、是否可写等状态。

而这个 `GenericNamedPropertyQueryCallback` 就返回这样的标识, 表示一个属性的状态。

下面先看一下 `EnvQuery` 的代码吧。

```
static void EnvQuery(Local<Name> property,
                    const PropertyCallbackInfo<Integer>& info) {
    int32_t rc = -1; // 除非另有说明, 否则标记找不到

    String::Value key(property);
    WCHAR* key_ptr = reinterpret_cast<WCHAR*>(*key);
    if (GetEnvironmentVariableW(key_ptr, nullptr, 0) > 0 ||
        GetLastError() == ERROR_SUCCESS) {
        rc = 0;
        if (key_ptr[0] == L'=') {
            // 环境变量若以 '=' 开头, 则其被视为隐藏以及只读
            rc = static_cast<int32_t>(v8::ReadOnly) |
                static_cast<int32_t>(v8::DontDelete) |
```

```

        static_cast<int32_t>(v8::DontEnum);
    }
}

if (rc != -1)
    info.GetReturnValue().Set(rc);
}

```

由于本片段代码的 Windows 相关逻辑更有解说价值，因此这里并没有用 UNIX 相关的逻辑。

老规矩，逐步解析代码：

- ① 声明结果变量，初始值为 -1。
- ② 获取 property 对象的底层真实字符串数据，赋给 key_ptr。
- ③ 进入分支条件：
 - (a) 如果获取到这个环境变量的值，则将结果变量赋值为 0，然后判断环境变量的键名第一位是不是等号 ("=")。
 - 若第一位是等号，则代表这是一个隐藏且只读的环境变量，那么将结果变量赋值为只读、不可删除以及不可枚举的 (v8::ReadOnly | v8::DontDelete | v8::DontEnum)。
 - 若第一位不是等号，则代表这是一个普通环境变量，将不做任何事。
 - (b) 获取不到环境变量的值，则不做任何事。
- ④ 最后判断结果变量是不是仍为初始值 -1。
 - (a) 若不是 -1，那么把拿到的结果（无论是普通的 0 还是经过加工的只读、不可删除以及不可枚举的）通过 info.GetReturnValue().Set 返回。
 - (b) 若是 -1，则代表根本没有该环境变量，那么直接不返回任何内容。

从代码解析中我们不难明白，这个函数就是告知 Chrome V8 这个拦截器中的某个属性是否存在，以及是否只读、可删除和可枚举。

这个状态的最终结果都是与 PropertyAttribute 这个枚举一一对应的，如表 3-4 所示。

表 3-4 PropertyAttribute 的枚举值

枚举名	值
None	0
ReadOnly	1
DontEnum	2
DontDelete	4

• GenericNamedPropertyDeleterCallback

这个函数是从对象拦截器中用于删除某个属性的回调函数。下面还是来看看 `process.env` 的相关代码。

```
static void EnvDeleter(Local<Name> property,
                      const PropertyCallbackInfo<Boolean>& info) {
    node::Utf8Value key(info.GetIsolate(), property);
    unsetenv(*key);

    // process.env 不会有不可配置的属性，所以就像 TC39 删除操作符一样，总是返回 true
    info.GetReturnValue().Set(true);
}
```

基本上无须解释了，看了前面几个函数之后大家应该都轻车熟路了。通过调用系统 API `unsetenv` 来将这个环境变量删除。

由于 `process.env` 中不存在不可重新配置的属性，因此删除肯定成功。这就是函数最后执行 `info.GetReturnValue().Set` 的时候始终为 `true` 的原因了。

• GenericNamedPropertyEnumeratorCallback

`GenericNamedPropertyEnumeratorCallback` 这个函数用于获取一个对象的枚举。对于这个定义不是很明晰的读者可以看如下这个例子。

```
// example.js
const obj = {
    foo: 1,
    bar: 2
};
Object.defineProperty(obj, "baz", {
    value: 3,
    enumerable: false
});

console.log(obj);
console.log(obj.baz);
for(const key in obj) {
    console.log(key);
}
```

这段代码的输出结果如下：

```
$ node example.js
{ foo: 1, bar: 2 }
3
foo
bar
```

因为我们在执行 `Object.defineProperty` 的时候插入了一个 "baz" 字段，令其值为 3，于是在执行 `console.log(obj.baz)` 的时候自然是输出 3 了。又因为我们定义这个 "baz" 字段的时候将其设为不可枚举的，因此它就不能在 `console.log` 中被枚举到并输出，于是第一次输出 `obj` 的时候就不存在 baz 这个字段。最后一个 `for...in` 循环就是对上面 `console.log` 的一个解释——因为这个 baz 并不会被 `for...in` 遍历到。

根据 ECMA 262 标准，在 `for...in` 的循环下，只会处理 `Enumerable` 为 `true` 的那些字段¹。

明白了前面说的这些之后，大家应该知道 `GenericNamedPropertyEnumeratorCallback` 的用意了——返回这个对象模板实例化后这个拦截器能被枚举到的字段名。

好了，现在看一下 Node.js 中的那个 `EnvEnumerator` 代码片段吧。

```
void EnvEnumerator(const PropertyCallbackInfo<Array>& info) {
  Environment* env = Environment::GetCurrent(info);
  Isolate* isolate = env->isolate();
  Local<Context> ctx = env->context();
  Local<Function> fn = env->push_values_to_array_function();
  Local<Value> argv[NODE_PUSH_VAL_TO_ARRAY_MAX];
  size_t idx = 0;

  int size = 0;
  while (environ[size])
    size++;

  Local<Array> envarr = Array::New(isolate);

  for (int i = 0; i < size; ++i) {
    const char* var = environ[i];
    const char* s = strchr(var, '=');
    const int length = s ? s - var : strlen(var);
    argv[idx] = String::NewFromUtf8(isolate,
                                     var,
                                     String::kNormalString,
```

¹ 该条规范在 ECMA 262 标准的 12.6.4 节 “The for-in Statement” 中，读者可自行翻阅 <https://www.ecma-international.org/ecma-262/5.1/#sec-12.6.4>。

```

                                length);
    if (++idx >= arraysize(argv)) {
        fn->Call(ctx, envarr, idx, argv).ToLocalChecked();
        idx = 0;
    }
}
if (idx > 0) {
    fn->Call(ctx, envarr, idx, argv).ToLocalChecked();
}

info.GetReturnValue().Set(envarr);
}

```

继续逐步解析。

- ① 声明一堆需要用到的变量（其中 `size` 指的是环境变量的数量）。
- ② 开始从 0 遍历环境变量数组 `environ` 得到各种环境变量，在每个循环中做如下操作。
 - (a) 将获取到的环境变量通过 "=" 分割，得到键值字符串和键名字符串的长度。
 - (b) 根据键名字符串的长度去截取环境变量的整个字符串，得到键名，然后通过 `String::NewFromUtf8` 生成 Chrome V8 所用的字符串，并加入缓冲区 `argv`。
 - (c) 进入缓冲区条件分支：
 - 若缓冲区满了，则将缓冲区的结果推入这个函数的结果变量 `Local<Array>` `envarr` 中，并清空缓冲区。
 - 若缓冲区没满则不做什么事。
- ③ 遍历结束后再检查一遍缓冲区是否有未推入结果的内容，若有则最后推一次。
- ④ 把包含这些环境变量键名的数组本地句柄 `envarr` 设置到 `info.GetReturnValue()`，准备返回给下游。

用一句话概括，就是获取环境变量，然后拿到这些环境变量的键名组成一个 Chrome V8 中的数组类型数据，最后返回。

• 自己写一个映射型拦截器

大家如果感兴趣的话，也可以自己尝试着写一个映射型拦截器，或者跟随随书代码“6. mapped property interceptor”的脚步进行学习——打开 `interceptor.cpp`。

首先，为了编写代码方便，HTTP 请求的相应代码直接使用了网上一个开源的简易 HTTP 请求器¹，配合 mbed TLS² 来对 HTTPS 请求进行发送和接收结果。大家不要太过拘泥于细节——开源库本来就是拿来用的。

其次，这段代码只是为了便于理解而写的一段样例，极度不推荐读者用于自己的生产环境。原因有如下几点：

- ① 有 HTTP 网络请求，且是同步操作，会阻塞进程。
- ② 没有容错性。
- ③ 在 getter 等函数中都会触发一次网络请求，那么在 for...in 中就会有几十次串行同步的网络请求。

明白了这几点之后，我们就开始解析代码吧。

首先是扩展加载的入口函数。

```
void Init(Local<Object> exports, Local<Object> module)
{
    minihttp::InitNetwork();
    atexit(minihttp::StopNetwork);

    Isolate* isolate = Isolate::GetCurrent();
    HandleScope scope(isolate);

    Local<ObjectTemplate> tpl = ObjectTemplate::New(isolate);
    tpl->SetHandler(v8::NamedPropertyHandlerConfiguration(
        Getter,
        0,
        Query,
        0,
        Enumerator,
        Local<Value>(),
        PropertyHandlerFlags::kOnlyInterceptStrings));

    module->Set(
        String::NewFromUtf8(isolate, "exports"),
        ((v8::MaybeLocal<Object>)tpl->NewInstance()).ToLocalChecked());
}
```

1 该开源的简易 HTTP 请求器被称为 minihttp，仓库地址是 <https://github.com/fgenesis/minihttp>，系笔者在 GitHub 上找的。

2 mbed TLS 的前身是 PolarSSL，这是一套小巧的 SSL 代码库，其高效、便于移植和集成。在本书中用作 minihttp 的一个依赖，用于对 HTTPS 请求的发起，其官方网站是 <https://tls.mbed.org/>。

当用户在其他文件通过 `require` 载入这个扩展的时候，`minihttp` 会先进行一次初始化。然后会生成一个对象模板，紧接着这个对象模板被装上拦截器，最后将通过这个对象模板实例化出来的对象挂载到 `module.exports` 下。

由于篇幅限制，这个对象模板的拦截器就实现了 `Getter`、`Query` 和 `Enumerator`，且其间的所有属性都是只读、可枚举、不可删除的。

接下来看看 `Enumerator` 函数——当对象需要被枚举的时候所使用的函数。

```
Local<Array> GetList(Isolate* isolate)
{
    EscapableHandleScope scope(isolate);

    // 访问 CNode.js API
    char* content = minihttp::Download("https://cnodejs.org/api/v1/topics");

    // 将得到的结果用 v8::JSON::Parse 进行 JSON 字符串解析，得到一个 v8 数据的
    // MaybeLocal 句柄
    MaybeLocal<Value> maybe = v8::JSON::Parse(isolate,
String::NewFromUtf8(isolate, content));

    // 将 MaybeLocal 句柄本地化
    Local<Value> ret = maybe.ToLocalChecked();

    // 释放 `content`
    free(content);

    // 取 JSON 对象的 `data` 字段，得到 `data` 数组
    //
    // 根据 CNode 社区的 API 文档，获取帖子列表 API 的 `data` 内容是包含一堆帖子对象的数组
    Local<Array> data = Local<Array>::Cast(ret->ToObject()->Get(String::NewFromUtf8(isolate, "data"))->ToObject());

    printf("fetching %s for %s...ok\n", "https://cnodejs.org/api/v1/topics",
"list");

    return scope.Escape(data);
}

void Enumerator(const PropertyCallbackInfo<Array>& info)
{
    Isolate* isolate = info.GetIsolate();
    HandleScope scope(isolate);

    // 获取帖子数组
```

```

Local<Array> data = GetList(isolate);

...

// 遍历帖子数组
for(unsigned int i = 0; i < data->Length(); i++)
{
    // 拿到第 i 个帖子的对象
    Local<Object> element = data->Get(Number::New(isolate, i))-
>ToObject();

    // 拿到第 i 个帖子的 `id` 值
    Local<String> id = element->Get(String::NewFromUtf8(isolate, "id"))-
>ToString();

    // 帖子数组的第 i 个数据用 `id` 值替换
    data->Set(Number::New(isolate, i), id);
}

// 最后得到一个包含一堆帖子 `id` 的数组,
// 通过 info.GetReturnValue() 返回
info.GetReturnValue().Set(data);
}

```

从代码中看到，当这个对象需要被枚举（如 `for...in`）的时候，扩展将执行 `GetList` 函数获取数组，然后将数组中的所有元素都用该元素中的 `id` 字段替换，最后将这个 `id` 的数组返回。

在 `GetList` 中，我们访问了 CNode 社区¹ 的开放 API，获取第一页帖子的数据并返回。这代表了什么呢？我们看看执行结果就知道了。进入“6. mapped property interceptor”目录，执行 `node-gyp rebuild` 之后再执行 `node`。

```

$ node-gyp rebuild
...
$ node
> const a = require("../build/Release/mapped_property_interceptor");
undefined
> Object.keys(a);
[ '58eee565a92d341e48cfe7fc',
  '58e607b0ddee72813eb22323',
  '58d0fb3517f61387400b7e15',

```

¹ CNode 社区 (<https://cnodejs.org>) 为国内最专业的 Node.js 开源技术社区，其致力于 Node.js 的技术研究，笔者也是该社区的管理员之一。关于 CNode 社区的 API 文档可以访问 <https://cnodejs.org/api> 查看。

```
'58ad76db7872ea0864fedfcc',
'588463a9250bf4e2390e9ea2',
'5919222f9e32cc84569a6f89',
'59197a509e32cc84569a6fb6',
'59187c62d371b6372a8af9ae',
'591135c3ba8670562a40ee47',
'591162809e32cc84569a6cbd',
'5897f95cf46268be08aea4fb',
'584a18289ff0dbf333450901',
... ]
```

当然，由于 CNode 第一页帖子的时效性，读者执行的时候可能会和本书中的数据不一样，但是意义是一样的——当前时间的第一页 CNode 社区的帖子 id 数组。

有兴趣的读者可以使用 <https://cnodejs.org/api/v1/topics> 这个 API 在浏览器中打开验证一下。

下面在介绍两个拦截器回调前，先介绍一下 GetTopic 函数，因为 Getter 和 Query 都会用到这个函数。

```
Local<Object> GetTopic(Isolate* isolate, const char* id, const char* usage)
{
    EscapableHandleScope scope(isolate);

    // 将基网址与 `id` 拼接
    // 形成类似于 https://cnodejs.org/api/v1/topic/54194f9a3ec6258b6420831f 的地址
    int url_length = strlen("https://cnodejs.org/api/v1/topic/") +
strlen(id);
    char url[url_length + 1];
    url[0] = 0;
    strcat(url, "https://cnodejs.org/api/v1/topic/");
    strcat(url, id);

    // 为了让读者在执行的过程中更好理解
    // 这里输出了一点内容
    printf("fetching %s for %s...", url, usage);

    // 访问 CNode.js API
    char* content = minihttp::Download(url);

    // 如果无法获取到内容，则返回一个空句柄
    //
    // 当某个 id 的帖子不存在的时候，CNode 会返回非 200 状态
    // minihttp 库在非 200 状态返回的时候，`content` 就会为空
    if(0 == content)
    {
```

```

        printf("failed\n");
        return Local<Object>();
    }

    // 如果有内容, 就将其用 JSON 解析成 V8 中的数据对象
    MaybeLocal<Value> maybe = v8::JSON::Parse(isolate,
String::NewFromUtf8(isolate, content));
    Local<Value> ret = maybe.ToLocalChecked();

    // 释放 content
    free(content);

    // 如果结果的 "success" 字段为 `false`, 则返回空句柄
    if(ret->ToObject()->Get(String::NewFromUtf8(isolate, "success"))->
ToBoolean()->IsFalse()) {
        printf("failed\n");
        return Local<Object>();
    }

    // 得到 `data` 数据, 即指定 id 的帖子 JSON 数据
    Local<Object> data = ret->ToObject()->Get(String::NewFromUtf8(isolate,
"data"))->ToObject();

    // 为了让读者在执行的过程中更好理解
    // 这里输出一部分内容
    printf("ok\n");

    // 通过可逃句柄作用域返回帖子的 JSON 数据
    return scope.Escape(data);
}

```

结合源码和注释, 我们很快就会明白, 这个函数用于根据指定帖子的 id 返回帖子内容的 JSON 数据。

接下来我们就来看看 Getter 函数吧。

```

void Getter(Local<Name> property, const PropertyCallbackInfo<Value>& info)
{
    Isolate* isolate = info.GetIsolate();
    HandleScope scope(isolate);

    // 获取这次拦截器访问的 `key`
    //
    // 在 Enumerator 函数中, 这个 `key` 就代表帖子的 id
    String::Utf8Value key(property);

```

```
// 往 `GetTopic` 传入帖子的 id (也就是 key), 然后得到帖子的数据
Local<Object> data = GetTopic(isolate, *key, "getter");

...

// 如果帖子正常获取
if(!data.IsEmpty())
{
    // 返回帖子的标题
    info.GetReturnValue().Set(data->Get(String::NewFromUtf8(isolate,
"title"))->ToString());
}
}
```

在 `Getter` 函数中, 我们拿到进来访问的键名, 即帖子的 `id`, 然后拿着帖子的 `id` 去 `GetTopic` 请求获取帖子的数据。最后把帖子的数据中的 `title` 值返回。

综上所述, 通过这个拦截器生成的对象内容是一个 帖子 `id` => 帖子标题 的键值对对象, 如:

```
{ '54194f9a3ec6258b6420831f': ' 图片主题色提取算法小结 (Node.js 版) ',
  '58eee565a92d341e48cfe7fc': '2017, 我们来聊聊 Node.js',
  '58e607b0ddee72813eb22323': 'cnpm@5 beta 测试招募 ',
  '59197a509e32cc84569a6fb6': ' 浏览器禁用了 js 怎么办 ',
  '59187c62d371b6372a8af9ae': 'Express 如何判断用户所处地区? ',
  '591135c3ba8670562a40ee47': 'ngnix 如何通过二级目录代理多个 nodejs 应用? ',
  '591162809e32cc84569a6cbd': 'nodejs 里有没有可以存储 callback 函数的缓冲队列?',
  '5897f95cf46268be08aea4fb': 'ubunut 系统下 crontab 定时 操作 pm2 无效, 应该怎么玩 ',
  '584a18289ff0dbf333450901': '简单高效的 nodejs 爬虫模型 ',
  ... }
```

最后就是 `Query` 函数了, 返回各属性的状态。让我们先来规划一下这个函数要怎么写吧。

- ① 拿到拦截器访问进来的 `key`, 也就是帖子的 `id`。
- ② 拿着帖子的 `id` 去请求帖子的 `JSON` 数据。
 - (a) 若数据正常, 则说明有这个字段 (属性), 把所有存在的属性都设置为只读、不可删、可枚举的。
 - (b) 若数据不正常, 则不做任何事。

瞧吧, 不用看代码大家也都能规划得出来。那么开始动手吧。

```
void Query(Local<Name> property, const PropertyCallbackInfo<Integer>& info)
{
```

```

Isolate* isolate = info.GetIsolate();
HandleScope scope(isolate);

String::Utf8Value key(property);

Local<Object> data = GetTopic(isolate, *key, "query");

...

if(!data.IsEmpty())
{
    info.GetReturnValue().Set(
        static_cast<int32_t>(v8::ReadOnly) |
        static_cast<int32_t>(v8::DontEnum));
    return;
}
}

```

事已至此，大家是不是都跃跃欲试了呢？在大家编译执行之前，笔者在这里提醒一句，还记得 `GetTopic` 函数里面的几个 `printf` 吗？它在执行的时候可很有用呢。

快上车，来不及解释了。等执行完之后笔者再慢慢解释吧。不过执行的时候，由于有大量的网络请求，还是同组阻塞的，因此结果会慢慢地一条一条出，请耐心等待。

```

$ node-gyp
...
$ node
> const a = require("../build/Release/mapped_property_interceptor");
undefined
> a
fetching https://cnodejs.org/api/v1/topic/inspect for getter...failed
fetching https://cnodejs.org/api/v1/topics for list...ok
fetching https://cnodejs.org/api/v1/topic/valueOf for getter...failed
fetching https://cnodejs.org/api/v1/topic/constructor for query...failed
fetching https://cnodejs.org/api/v1/topic/58eee565a92d341e48cfe7fc for
query...ok
fetching https://cnodejs.org/api/v1/topic/58eee565a92d341e48cfe7fc for
getter...ok
fetching https://cnodejs.org/api/v1/topic/58e607b0ddee72813eb22323 for
query...ok
fetching https://cnodejs.org/api/v1/topic/58e607b0ddee72813eb22323 for
getter...ok
fetching https://cnodejs.org/api/v1/topic/58d0fb3517f61387400b7e15 for
query...ok
fetching https://cnodejs.org/api/v1/topic/58d0fb3517f61387400b7e15 for

```

```

getter...ok
fetching https://cnodejs.org/api/v1/topic/58ad76db7872ea0864fedfcc for
query...ok
fetching https://cnodejs.org/api/v1/topic/58ad76db7872ea0864fedfcc for
getter...ok
...
{ '58eee565a92d341e48cfe7fc': '2017, 我们来聊聊 Node.js',
  '58e607b0dde72813eb22323': 'cnpm@5 beta 测试招募 ',
  '59187c62d371b6372a8af9ae': 'Express 如何判断用户所处地区? ',
  '591135c3ba8670562a40ee47': 'nginx 如何通过二级目录代理多个 nodejs 应用? ',
  '591162809e32cc84569a6cbd': 'nodejs 里有没有可以存储 callback 函数的缓冲队列?',
  '584a18289ff0dbf333450901': '简单高效的 nodejs 爬虫模型 ',
  '5915d78c3504ce1c2ac45bcb': '朴灵大大的书好难看懂 ',
  '5829631dd3abab717d8b4c2c': '用 Node 和 Express 打造 restful API',
  '591962219e32cc84569a6fa4': 'webstorm 无法正确检查 ejs 模板的 if-else 语句 ',
  '591943e79e32cc84569a6f8f': 'express-session 怎么支持无 cookie 的微信小程序
session',
  '55a4b2123ecc81b621bba8c7': 'Node.js 挖掘之三: 一张图介绍 Node.js 的各路英雄 ',
  '591130d6ba8670562a40ee44': '从深入浅出 Node.js 中入门可以吗? ',
  '591918cd9e32cc84569a6f78': '新手, 想咨询下 node 下 ftp 上传问题 ',
  '54194f9a3ec6258b6420831f': '图片主题色提取算法小结 (Node.js 版)',
  '5918fc27ba8670562a40f101': '谁有 现在主流的各个 RESTful 框架 的性能对比分析? ',
  ... }

```

由于篇幅太长，因此这里只截取运行结果的前部分内容。

第一步把 `mapped_property_interceptor` 这个扩展先加载进来。然后单独执行一个 `a`，Node.js 就会输出这个 `a`。如果大家不是很理解，换成 `console.log(a)` 的结果也是差不多的。

既然是输出 `a`，那肯定是免不了遍历的。首先，Chrome V8 会尝试访问 `a.inspect()` 函数，这个函数和 `toString()` 差不多，会直接返回它的结果。

Inspect

我们来看看下面这段代码：

```

const obj = { foo: 1 };
console.log(obj);
obj.inspect = function() {
  return "覆盖输出";
};
console.log(obj);

```

执行这样的代码之后，会发现有两次输出：

① 第一次输出 `obj` 的结果是 `{ foo: 1 }`；

② 第二次输出 `obj` 的结果是 `覆盖输出`。

这说明一个 JavaScript 的值在没有 inspect 函数的时候，该怎么输出就这么输出；一旦有了 inspect，则会直接调用 inspect 函数进行输出。

在我们知道 inspect 这个函数的作用之后，输出的第一句是 fetching https://cnodejs.org/api/v1/topic/inspect for getter...failed 就在预料之中了。因为 Chrome V8 会先去获取这个对象的 inspect 属性，于是走到了 getter 并且访问的键值是 "inspect"。最后经过获取之后发现并没有这个属性，那么程序就继续往下走了。

接下去的步骤就是去获取这个对象的键名枚举了。也就是说会通过 Enumerator 函数——这才有了第二句输出 fetching https://cnodejs.org/api/v1/topics for list...ok。

那么接下去的第三句输出 fetching https://cnodejs.org/api/v1/topic/valueOf for getter...failed 和第四句输出 fetching https://cnodejs.org/api/v1/topic/constructor for query...failed 是什么呢？

别急，让我们再来看一段 JavaScript 代码：

```
// 为了更有利于理解，这里使用原型链，而不是 `class`

const TestObject1 = function TestObject1() {
  this.a = 2333;
  this.b = 233;
};

const obj1 = new TestObject1();
console.log(obj1);

const TestObject2 = function TestObject2() {
  this.a = 2333;
  this.b = 233;
};

TestObject2.prototype.valueOf = function() {
  return this.a + this.b;
};

const obj2 = new TestObject2();
console.log(obj2);
```

这段代码的两次 console.log 输出结果如下：

```
TestObject1 { a: 2333, b: 233 }
{ [Number: 2566] a: 2333, b: 233 }
```

在代码的前半段, `TestObject1` 这条原型链的构造函数是 `function TestObject1() { ... }`, 这相当于 `obj1.constructor` 就是这个 `TestObject1` 函数。因为 `obj1` 有构造函数(即 `obj1.constructor`), 所以在输出这个对象的时候会在前面加上构造函数名, 表示这是一个 `TestObject1` 类的实例。

在后半段代码中, 虽然 `obj2` 也有构造函数, 不过它还有一个 `valueOf` 函数, 代表这个对象在运算时用这个值作为元值来运算。所以在输出时, **Chrome V8** 在对象头加上了 `valueOf` 的结果来取代构造函数名。

理解了这样一个输出顺序之后, 我们映射型拦截器样例中的代码输出顺序问题也就迎刃而解了。

继续回到样例代码输出的第三句, **Chrome V8** 尝试获取 `valueOf` 来支持输出, 所以才有了使用 `valueOf` 进入 `getter` 这个步骤。但是发现这个对象并没有 `valueOf`, 于是放弃了。接下去它还不死心, 接下去尝试获取 `constructor` 来支持输出, 于是有了第四句的输出——很遗憾, 这个对象仍然没有 `constructor`。

接下去的输出就好理解了, 根据枚举函数返回的数组, 通过 `Query` 函数逐一去访问这些属性的状态。由于这里面的逻辑是我们自己写的, 这些属性的状态显然都是只读、不可删、可枚举的。**Chrome V8** 发现这个属性是可枚举的, 于是它继续通过 `getter` 去访问这些属性的实体, 得到属性实体为这些帖子的 `title`。在这个过程中, 理所当然就具有了同一个 API 出现 `query` 和 `getter` 交替的状况了。

当整个遍历完成下来之后, **Chrome V8** 就得到了 `id => title` 的键值对对象了。于是这个对象的本体就被输出了。

你以为事情就这样结束了吗? 没有, 笔者还能在上面的操作中继续做点好玩的 Hack。

```
$ node
> const a = require("../build/Release/mapped_property_interceptor");
undefined
> a.constructor = function Test() {};
[Function: Test]
> a
...
Test {
  constructor: [Function: Test],
  '58eee565a92d341e48cfe7fc': '2017, 我们来聊聊 Node.js',
  '58e607b0dde72813eb22323': 'cnpn@5 beta 测试招募 ',
  '59187c62d371b6372a8af9ae': 'Express 如何判断用户所处地区? ',
  ... }
```

```

> a.valueOf = function() { return "被我覆盖啦"; };
[Function]
> a
...
{ [String: '被我覆盖啦']
  constructor: [Function: Test],
  valueOf: [Function],
  constructor: [Function: Test],
  '58eee565a92d341e48cfe7fc': '2017, 我们来聊聊 Node.js',
  '58e607b0ddee72813eb22323': 'cnpm@5 beta 测试招募',
  '59187c62d371b6372a8af9ae': 'Express 如何判断用户所处地区? ',
  ... }

```

这样操作的结果是不是都在你的意料之中呢？

最后言归正传，笔者再直接试验一下 getter 的效果，而不是间接地输出整个对象。

```

$ node
> const a = require("../build/Release/mapped_property_interceptor");
undefined
> a["54194f9a3ec6258b6420831f"];
'图片主题色提取算法小结 (Node.js 版)'

```

我们能发现，不管我们通过 `a["帖子 id"]` 传入的 `id` 是否在枚举回调时被返回过，只要是一个有效的帖子 `id`，它总能出结果。原因在于，当我们直接通过一个属性名去访问对象属性的时候，它并不需要经过枚举回调验证。

（2）索引型拦截器（Indexed Property Interceptor）

索引型拦截器拦截的是数字下标访问的属性。我们可以用 JavaScript 中普通对象和数组的不同来类比映射型拦截器和索引型拦截器的不同。

与映射型拦截器不同的是，就目前而言，Node.js 并没有使用索引型拦截器。不过那又怎么样呢？这不妨碍读者学习索引型拦截器的热情啊。

就使用方法来说，索引型拦截器与映射型拦截器大同小异。它们均通过 `SetHandler` 函数来设置拦截器，只不过索引型拦截器传的参数是一个 `IndexedPropertyHandlerConfiguration` 的对象，这个类的构造函数如下：

```

IndexedPropertyHandlerConfiguration::IndexedPropertyHandlerConfiguration(
    IndexedPropertyGetterCallback getter = 0,
    IndexedPropertySetterCallback setter = 0,
    IndexedPropertyQueryCallback query = 0,
    IndexedPropertyDeleterCallback deleter = 0,

```

```
IndexedPropertyDescriptorCallback enumerator = 0,
Local<Value> data = Local<Value>(),
PropertyHandlerFlags flags = PropertyHandlerFlags::kNone);
```

这看起来也与映射型拦截器的配置对象基本一致，只不过里面的各种回调函数的类型前缀不一样。当然，写这些函数的时候也是不一样的。

在映射型拦截器的各种回调函数中，第一个参数是一个 Name 数据对象的本地句柄（Local<Name>）；索引型拦截器的各种回调函数中的第一个参数则“直勾勾”地是一个 uint32_t 型的 C++ 底层数据，也就是无符号 32 位整型，即 unsigned int。

要不要继续尝试着用索引型拦截器去 CNode 社区拿数据呢？

由于其与映射型拦截器非常相像，因此本书就不再赘述了。读者直接看看代码片段琢磨吧。大家可以打开随书代码“7. indexed property interpector”的 interpector.cpp。

```
Local<Array> GetList(Isolate* isolate)
{
    EscapableHandleScope scope(isolate);

    // 访问 CNode.js API
    char* content = minihttp::Download("https://cnodejs.org/api/v1/topics");
    MaybeLocal<Value> maybe = v8::JSON::Parse(isolate,
String::NewFromUtf8(isolate, content));
    Local<Value> ret = maybe.ToLocalChecked();
    free(content);

    // 得到 `data` 数组
    Local<Array> data = Local<Array>::Cast(ret->ToObject()->Get(String::NewFromUtf8(isolate, "data"))->ToObject());

    printf("fetching %s for %s...ok\n", "https://cnodejs.org/api/v1/topics",
"list");

    return scope.Escape(data);
}

void Getter(uint32_t index, const PropertyCallbackInfo<Value>& info)
{
    Isolate* isolate = info.GetIsolate();
    HandleScope scope(isolate);

    Local<Array> data = GetList(isolate);
```

```

...

if(!data.IsEmpty() && index < data->Length())
{
    info.GetReturnValue().Set(
        data->Get(Number::New(isolate, index))->ToObject()->Get(String::NewFromUtf8(isolate, "title")));
}
}

void Query(uint32_t index, const PropertyCallbackInfo<Integer>& info)
{
    Isolate* isolate = info.GetIsolate();
    HandleScope scope(isolate);

    Local<Array> data = GetList(isolate);

    ...

    if(!data.IsEmpty() && index < data->Length())
    {
        info.GetReturnValue().Set(
            static_cast<int32_t>(v8::ReadOnly |
                                static_cast<int32_t>(v8::DontEnum)));
        return;
    }
}

void Enumerator(const PropertyCallbackInfo<Array>& info)
{
    Isolate* isolate = info.GetIsolate();
    HandleScope scope(isolate);

    Local<Array> data = GetList(isolate);

    ...

    for(unsigned int i = 0; i < data->Length(); i++)
    {
        data->Set(Number::New(isolate, i), Number::New(isolate, i));
    }

    info.GetReturnValue().Set(data);
}

void Init(Local<Object> exports, Local<Object> module)

```

```

{
    minihttp::InitNetwork();
    atexit(minihttp::StopNetwork);

    Isolate* isolate = Isolate::GetCurrent();
    HandleScope scope(isolate);

    Local<ObjectTemplate> tpl = ObjectTemplate::New(isolate);
    tpl->SetHandler(v8::IndexedPropertyHandlerConfiguration(
        Getter,
        0,
        Query,
        0,
        Enumerator,
        Local<Value>(),
        PropertyHandlerFlags::kNone));

    module->Set(
        String::NewFromUtf8(isolate, "exports"),
        ((v8::MaybeLocal<Object>)tpl->NewInstance()).ToLocalChecked());
}

```

然后就能编译执行并看结果了。

```

$ node-gyp rebuild
...
$ node
> const a = require("./build/Release/indexed_property_interceptor");
undefined
> a
fetching https://cnodejs.org/api/v1/topics for list...ok
fetching https://cnodejs.org/api/v1/topics for list...ok
fetching https://cnodejs.org/api/v1/topics for list...ok
...
{ '0': 'cnpm@5 beta 测试招募 ',
  '1': '2017, 我们来聊聊 Node.js',
  ...
  '38': 'mongoose 中怎么查找某个字段的最值, (最大最小值)的数据, 网上没找到 ',
  '39': 'npm install express -g 全局安装 Express 后命令行输入 Express 提醒
"express 不是内部或外部命令的问题" ' }
> a[15]
' 图片主题色提取算法小结 (Node.js 版) '
> a[40]
undefined

```

从执行结果中我们可以看出，代码基本上按照我们的意愿执行了。不过有一点值得注意，虽然它在名称上叫索引型拦截器，并且之前笔者也是以对象和数组进行类比的，但是索引型拦截器服务的结果仍然是一个对象，只不过属性名是可转换为数字的字符串而已。

而且，访问超过总数的下标的结果是 `undefined`——因为这个样例的底层逻辑本身就不管不在当前数组里的内容，也不会去访问额外的 API 接口。

3.6.4 对象模板的内置字段（Internal Field）

在 V8 中，能与 JavaScript 代码中直接交互的数据类型都是以句柄形式出现的 V8 数据类型，如 `v8::Number`、`v8::String` 等，以及对象 `v8::Object`。在 `v8::Object` 中，存在的也都是些同类的数据类型。

如果我们有一个自身的底层数据结构，需要怎样跟 V8 的数据类型联系起来，或者说，如何绑在一起呢？

这里就涉及了 V8 对象的一个概念——内置字段。该字段对于 JavaScript 代码来说是不可见的，只有到 C++ 的层面，才能通过 `v8::Object` 的特定方法将其获取出来。读者可以通俗地将其理解为 V8 对象数据类型的私有属性。

所以总的来说，如果我们需要将一个自有数据结构的对象与一个 V8 对象联立起来，就通过一个方法将其设置为该对象的一个内置字段。不过在设置内置字段之前，需要设置一下这个对象所对应模板的内置字段数。

比如，我们有一个 `Person` 的结构体，其中有一些它自带的字段：

```
enum Gender {  
    MALE,  
    FEMALE  
};  
  
struct Person {  
    char name[256];  
    int age;  
    Gender gender;  
};
```

然后我们想将其藏入一个 V8 的对象中，就先要弄出一个对象模板，并将其内置字段数设置为 1。

```
Local<ObjectTemplate> templ = ObjectTemplate::New(isolate);
templ->SetInternalFieldCount(1);
```

在 Chrome V8 中，对象模板关于内置字段数的 API 有以下两个。

- `int ObjectTemplate::InternalFieldCount()`: 获取该模板的内置字段数。
- `void ObjectTemplate::SetInternalFieldCount(int value)`: 设置该模板的内置字段数。

这个内置字段数需要自身按需设置，建议不要滥用，如明明只需要一个内置字段，这里却偏偏设置了好多个。

在设置了内置字段数之后，我们就能为通过这个对象模板实例化出来的对象设置内置字段了。

```
Person* person = new Person();

// 对 person 做一些事情
...

Local<Object> obj = templ->NewInstance();
obj->SetInternalField(0, External::New(isolate, person));
```

V8 对象对于内置字段的常用 API 有以下 3 个。

- `int Object::InternalFieldCount()`: 获取这个对象的内置字段数。
- `Local<Value> Object::GetInternalField(int index)`: 获取这个对象被设置的第 `index` 个内置字段。
- `void Object::SetInternalField(int index, void* value)`: 设置这个对象的第 `index` 个内置字段。

细心的读者可能发现了，我们在藏 `person` 这个内置字段的时候，外面还通过 `External::New` 将其包了一层再传入 `SetInternalField`。

实际上，在 `Object` 中，传入的还得是一个 V8 的句柄——而这个句柄中才是真正藏内置字段的地方。而这个句柄对应的数据类型叫 `v8::External`，其同样继承自 `v8::Value`，也就是一种 V8 的数据类型。

当调用 `External::New` 这个静态函数的时候，就得到了一个 `External` 数据类型的本地句柄，其中隐藏了传入这个静态函数的指针。紧接着通过 `SetInternalField` 将这个本地 `External` 句柄绑定到对象中。

我们可以将得到的这个对象通过类似于模块初始化或者 C++ 函数模板等各种形式传入 JavaScript 层面，这样这个 External 对象也会随之而去。当这个对象重新被传入 C++ 层面的扩展时，我们就能通过 GetInternalField 获取这个被隐藏的字段，使其重见天日。

这个时候就该打开随书源码“8. internal field wrong”来看看样例了。本节前面的代码基本来自这个样例。

```
void Init(Local<Object> exports, Local<Object> module)
{
    Isolate* isolate = Isolate::GetCurrent();
    HandleScope scope(isolate);

    Local<ObjectTemplate> templ = ObjectTemplate::New(isolate);
    templ->SetInternalFieldCount(1);

    Person* person = new Person();

    strcpy(person->name, "芙兰朵露·丝卡蕾特");
    person->gender = Gender::FEMALE;
    person->age = 495;

    MaybeLocal<Object> dummy_obj = templ->NewInstance();
    Local<Object> obj = dummy_obj.ToLocalChecked();

    // 假设 obj 肯定不为空

    obj->SetInternalField(0, External::New(isolate, person));
    obj->Set((MaybeLocal<String>(String::NewFromUtf8(isolate,
    "getSummary"))).ToLocalChecked(),
            FunctionTemplate::New(isolate, GetSummary)->GetFunction());

    module->Set(String::NewFromUtf8(isolate, "exports"), obj);
}
```

在代码中可看到，首先笔者新建了一个 Person 指针，然后将其藏匿于 obj 对象中，顺手将一个函数 getSummary 也挂载到这个对象之下，最后通过 module.exports 将这个对象导出给使用者。

老规矩，执行以下代码看看效果：

```
$ node-gyp rebuild
...
$ node
> const obj = require("../build/Release/internal_field_wrong")
```

```
undefined
> obj
{ getSummary: [Function] }
```

其效果应该跟读者预期的差不多吧。当我们通过 `require` 载入这个写好的 C++ 扩展时，得到的是一个对象，而且这个对象中仅有一个 `getSummary` 函数。而那个 `person` 对象已经被深深隐藏了。

那么该怎么拿到这个 `person` 对象呢？大家可以尝试着把目光放到样例的 `GetSummary` 函数上去。

```
void GetSummary(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();
    Local<Object> self = args.Holder();
    Local<External> wrap = Local<External>::Cast(self->GetInternalField(0));

    Person* person = static_cast<Person*>(wrap->Value());

    char ret[512];
    sprintf(ret, "%s, %s, %d 岁。", person->name, person->gender == MALE ? "男" : "女", person->age);

    MaybeLocal<String> summary = String::NewFromUtf8(isolate, ret);

    args.GetReturnValue().Set(summary.ToLocalChecked());
}
```

下面逐步解析代码：

- ① 通过 `args.Holder()` 获取这个函数调用时的 `this`，因为我们使用的时候都是通过 `obj.getSummary()` 来调用的，所以在样例中实际上就是扩展 `exports` 得到的对象。
- ② 通过这个对象的 `GetInternalField` 函数获取第 0 个内置字段的本地句柄，也就是笔者之前通过 `External::New` 塞进去的 `Person` 包裹器。
- ③ 通过 `External` 对象的 `->Value()` 函数获取 `Person` 指针本体。
- ④ 拿着这个 `Person` 指针做一些计算，把结果放入 `ret` 中。
- ⑤ 将 `ret` 转成 V8 字符串并返回。

通过解析我们看到，这个函数用一句话总结就是，拿到这个对象的内置对象并做一些计算，将结果返回给 JavaScript 代码。

至此，我们基本上了解了对象模板和对象的内置字段的用法了。

思考：大家可以尝试着再多写几个函数，如通过 JavaScript 来设置内置对象的各值（如 name、age 等）。

不过值得注意的是，样例的代码没有做容错处理，所以当大家在调用的时候换一个 this 对象，就会使程序挂掉。读者可以尝试一下这样的命令：

```
$ node
> const obj = require("../build/Release/internal_field_wrong")
undefined
> obj
{ getSummary: [Function] }
> obj.getSummary.call({});
[1] 5820 segmentation fault node
```

当读者将调用 getSummary 函数的 this 用一个新对象替换时，Node.js 进程就崩溃了。因为这个时候新的对象是不存在内置字段的，所以在 GetInternalField 之后对这个莫须有的内置字段结果做操作就会导致 Node.js 崩溃。

思考：这里大家可以自行思考一下如何做容错。如在使用之前判断一下内置字段数是否吻合，以及在的确能获取到内置字段后通过一些手段判断一下拿到的指针是否是自己需要的内容等。

其实还有一个一刀切的办法，就是让你的使用者根本碰不到内置字段相关的对象，把这些对象深深地封装在你的上层 JavaScript 代码中，就跟 Node.js 自身的 lib 目录下的核心模块基本上都是基于 src 目录里面的 C++ 模块封装而成的一个道理。

大家可能以为关于内置字段的内容，笔者讲到这里就结束了。其实不然。细心的读者可能会发现，这一次样例代码的目录后缀跟着一个 wrong，那么是哪里“wrong”了呢？

写习惯了 JavaScript 的大家是否已经忘记了指针内存未回收而导致内存泄漏的恐惧了？

对，没错——这段代码里面没有任何一个地方对堆内存分配的 person 对象指针进行回收，那么这会导致什么结果呢？比如我们的一个对象如果不再使用，会被 V8 的垃圾回收机制回收，但是 External 对象里面的内置字段本体却不在 V8 的管辖范畴之内。在 V8 的对象再怎么回收，你自身实现的这些指针却永远不会被 V8 处理。

所以有一点非常重要，那就是读者必须自己掌控这些内置字段本体的生命周期。

但是，这谈何容易？一个内置字段本体如果回收早了，那么可能会导致下次需要访问这个内置字段时崩溃的情况；如果回收晚了，那可能就是内存泄漏。所以通常情况下（除非读者确

实想让它持续存在) 这个内置字段本体必须要与它的宿体 `External` 数据对象共存亡。换言之, 就是在本地 `External` 句柄被 V8 回收的时候做一些事情, 比如用 `delete` 将这个内置字段本体删除。

那么问题来了, 怎么知道这个本地 `External` 句柄什么时候被回收呢? 不知道大家还记得一套从天而降的“掌法”——在前面曾提到过的弱持久句柄。为什么这里会提到这么一个概念呢? 因为弱持久句柄的一个用处就是让你知道它什么时候被回收。读者回顾一下关于弱持久句柄的这么一句话:

当对一个 JavaScript 对象的引用只剩下一个弱持久句柄时, Chrome V8 的垃圾回收器就会触发一个回调。

也就是说, 一旦一个弱持久句柄指向的 V8 数据体的所有其他本地句柄引用或者其他类型的句柄引用全部都没有了, 它即将要被垃圾回收时, 这个回调就会被触发。

这不就是我们需要的 API 吗? 我们就是需要一个本地句柄所对应的 V8 数据体将要被回收时会被触发的回调函数。

明白了这一点, 再加上对照 3.3.2 节中关于弱持久句柄的一些 API 来看, 我们就可以解析“9. `internal field right`”的代码了。

首先看看导出到 `exports` 的函数 `create`。

```
struct PersistentWrapper {
    Persistent<Object> persistent;
    int value;
};

void CreateObject(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();

    // 新建对象模板
    Local<ObjectTemplate> templ = ObjectTemplate::New(isolate);
    templ->SetInternalFieldCount(1);

    // 新建对象以及设置内置字段
    PersistentWrapper* wrapper = new PersistentWrapper();
    wrapper->value = args[0]->ToNumber()->Int32Value();
    Local<Object> obj = templ->NewInstance();
    obj->SetInternalField(0, External::New(isolate, wrapper));

    // 基于 obj 新建持久句柄
```

```

wrapper->persistent.Reset(isolate, obj);

// 将持久句柄设置为弱持久句柄
wrapper->persistent.SetWeak(wrapper, WeakCallback, v8::WeakCallbackType:
:kInternalFields);
wrapper->persistent.MarkIndependent();

args.GetReturnValue().Set(obj);
}

```

根据代码中的注释，大家应该能很快弄明白这是怎么一回事。在这个函数返回的对象 `obj` 被 V8 进行垃圾回收时，就会触发一个 `WeakCallback` 回调，而在这个回调中就是删除笔者在这里 `new` 出来的 `PersistentWrapper` 了。

```

void WeakCallback(const v8::WeakCallbackInfo<PersistentWrapper>& data)
{
    PersistentWrapper* wrapper = data.GetParameter();

    if(!wrapper->persistent.IsNearDeath())
    {
        printf("WARN...\n");
    }

    printf("deleting 0x%.8X: %d...", wrapper, wrapper->value);
    wrapper->persistent.Reset();
    delete wrapper;
    printf("ok\n");
}

```

大家可以看到，在 `CreateObject` 函数中通过 `SetWeak` 传进来的 `wrapper` 能被 `WeakCallback` 里面的 `data.GetParameter()` 所获取。在获取之后，我们只需要将这个持久句柄重置，然后将堆内存分配出来的指针删除即可。为了演示方便，在样例代码中输出了一些调试信息。

接下来大家可以执行这里的代码了。

```

$ node-gyp rebuild
...
$ node
> const obj = require("../build/Release/internal_field_right");
undefined
> function test() { const ret = {}; for(let i = 0; i < 10; i++) ret[i] =
obj.create(i); return ret; }

```

```

undefined
> const ret = test();
undefined
> ret
{ '0': {},
  '1': {},
  '2': {},
  '3': {},
  '4': {},
  '5': {},
  '6': {},
  '7': {},
  '8': {},
  '9': {} }
> delete ret[0]
true
> delete ret[1]
true
> delete ret[2]
true
> delete ret[3]
true
> delete ret[4]
true

...
deleting 0x03400190: 0...ok
deleting 0x03400050: 1...ok
deleting 0x03400060: 2...ok
deleting 0x034001E0: 3...ok
deleting 0x034001F0: 4...ok

```

我们可以看到,在上面的 `test()` 函数中创建了 10 个通过 `CreateObject` 创建出来的对象,这些对象都与其各自的 `WeakCallback` 关联起来。

然后我们逐个通过 `delete` 删除返回对象里面的各个属性。删除到一定程度(具体删除到什么时候不一定)时,V8 的垃圾回收机制有可能会执行。这时被回收的那些对象的弱持久句柄的回调也会被触发,于是我们就能看到笔者事先预留好的调试信息了。

当然这样调试需要看运气,要等到 Chrome V8 的垃圾回收机制被触发才行。笔者还有更简单、更直接的调试验证方法。

```

$ node
> const obj = require("../build/Release/internal_field_right")

```

```

undefined
> function test() { for(let i = 0; i < 10; i++) obj.create(i); }
undefined
> test()
undefined

...
deleting 0x01F0A6B0: 0...ok
deleting 0x01F0ACE0: 1...ok
deleting 0x01F0B970: 2...ok
deleting 0x01F0B980: 3...ok
deleting 0x01F0B5B0: 4...ok
deleting 0x01F0B5C0: 5...ok
deleting 0x01F0B5D0: 6...ok
deleting 0x01F0B5E0: 7...ok
deleting 0x01F0B3F0: 8...ok
deleting 0x01F0B400: 9...ok

```

这下大家明白了一个内置字段的使用姿势了吧。实际上，这种姿势并不是笔者臆想出来的，而是参照了 Node.js 的 `ObjectWrapper` 类。

Node.js 自身为了让 C++ 的开发者更方便地对内置字段进行操作，基于上面的一套姿势封装了一个叫作 `ObjectWrapper` 的类，有兴趣的读者可以像笔者一样去研究研究它（https://github.com/nodejs/node/blob/v6.9.4/src/node_object_wrap.h）。而 `ObjectWrapper` 这个类自身的用法也会在 4.3.1 节中被提及。

思考：大家可以根据“9. internal field right”中的姿势尝试把“8. internal field wrong”中的代码改成正确的姿势。

3.6.5 函数模板的继承（Inherit）

我们知道，JavaScript 在 ECMAScript 6 之前，是第一个不基于类的面向对象编程语言。所以说在 ECMAScript 6 之前，原型和原型链是它的精髓，对象的各种继承就是基于原型链来做到的。

在 3.6.1 节中，笔者曾提到过 `PrototypeTemplate()`，它返回的是这个函数模板的原型链上的原型对象模板。

下面看一个 *Embedder's Guide* 上的例子。

```

Local<FunctionTemplate> biketemplate = FunctionTemplate::New(isolate);
biketemplate->PrototypeTemplate().Set(
    String::NewFromUtf8(isolate, "wheels"),

```

```
FunctionTemplate::New(isolate, MyWheelsMethodCallback)->GetFunction();
);
```

这段代码就是给 `bicktemplate` 这个构造函数的原型链加上了一个 `wheels` 的函数。`MyWheelsMethodCallback` 的具体实现我们并不关心，这里就不展示了。

到此为止，仍停留在原型链赋值的概念上。

其实 V8 的函数模板还提供了一个函数 `Inherit()`。这个函数可以将一个函数模板继承自另一个函数模板。该函数原型如下：

```
void FunctionTemplate::Inherit(Local<FunctionTemplate> parent);
```

想象一下我们有两个类，宠物（`Pet`）和狗（`Dog`），让我们尝试用最简单的抽象去描述它们的继承关系。

```
const util = require("util");

function Pet() {
  this.name = "Unknown";
  this.type = "animal";
}

Pet.prototype.summary = function() {
  return `${this.name} is a/an ${this.type}.`;
}

Pet.prototype.setName = function(name) {
  this.name = name;
}

function Dog() {
  Pet.call(this);
  this.type = "dog";
}

util.inherit(Dog, Pet);
```


在 `Pet` 的构造函数中定义了对象的名字和类型，并给这个类（原型）加上 `summary` 和 `setName` 两个函数。`Dog` 通过 Node.js 中的 `util.inherit` 函数¹ 进行继承，并将自身的 `type` 改成 "dog"。这样一来，我们就能得到两个继承关系的类——父类 `Pet` 和子类 `Dog` 了。

当然，笔者不会就为了让大家看看上面的代码而放出来这些东西。上面这些无非是为下面要讲的 Chrome V8 中函数模板的继承样例而准备的。

请读者打开“新世界”的大门吧——“10. function template inherit”的 `inherit.cpp`，笔者会逐一讲解。

首先是 `Pet` 的构造函数。

```
void Pet(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();

    // 获取 `this`
    Local<Object> self = args.Holder();

    // this.name = "Unknown";
    // this.type = "animal";
    self->Set(String::NewFromUtf8(isolate, "name"),
String::NewFromUtf8(isolate, "Unknown"));
    self->Set(String::NewFromUtf8(isolate, "type"),
String::NewFromUtf8(isolate, "animal"));

    // 构造函数中需要将 `this` 返回，以供
    //
    //   const foo = new Foo();
    //
    // 的左值接收
    args.GetReturnValue().Set(self);
}

...

void Init(Local<Object> exports, Local<Object> module)
{
    ...
    Local<FunctionTemplate> pet = FunctionTemplate::New(isolate, Pet);
    ...
}
```

¹ 在 Node.js 出来之后，用该函数的封装对原型进行继承操作，相较于传统方法会更为方便和直观。这个函数的文档地址是 https://nodejs.org/docs/v6.9.4/api/util.html#util_util_inherits_constructor_superconstructor。

```

Local<Function> pet_cons = pet->GetFunction();

exports->Set(String::NewFromUtf8(isolate, "Pet"), pet_cons);
}

```

相信大家对这段代码已经熟悉得不能再熟悉了，全是之前讲过的各种姿势。应该也不需要笔者过多解释了吧。

然后为这个 `Pet` 的函数模板加上两个原型链上的函数。

```

Persistent<Function> cons;

void SetName(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();

    Local<Object> self = args.Holder();
    self->Set(String::NewFromUtf8(isolate, "name"), args[0]);
}

void Summary(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();

    Local<Object> self = args.Holder();
    char temp[512];

    String::Utf8Value type(self->Get(String::NewFromUtf8(isolate, "type"))-
>ToString());
    String::Utf8Value name(self->Get(String::NewFromUtf8(isolate, "name"))-
>ToString());

    snprintf(temp, 511, "%s is a/an %s.", *name, *type);

    args.GetReturnValue().Set(String::NewFromUtf8(isolate, temp));
}

void Init(Local<Object> exports, Local<Object> module)
{
    ...

    pet->PrototypeTemplate()->Set(
        String::NewFromUtf8(isolate, "setName"),
        FunctionTemplate::New(isolate, SetName));
    pet->PrototypeTemplate()->Set(

```

```

        String::NewFromUtf8(isolate, "summary"),
        FunctionTemplate::New(isolate, Summary));

    Local<Function> pet_cons = pet->GetFunction();

    cons.Reset(isolate, pet_cons);

    ...
}

```

上面的代码也不用过多解释了，也是先前章节中的各种知识点。不过有一个地方需要强调一下，笔者在代码中声明了一个 `Persistent<Function> cons` 的持久函数句柄，并在 `Local<Function> pet_cons` 这个构造函数生成的时候将它升格成一个持久句柄，这么做是为了在其子类 `Dog` 的构造函数中能使用它来进行一个函数调用，而这个函数类似于前面 JavaScript 源码中的 `Pet.call(this)`。这么说可能有些读者还是不太明白，不过当大家看到下面对于 `Dog` 构造函数的操作应该就会清楚了。

```

void Dog(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();

    // 获取 `this` 和 `Pet` 这个 `super`
    Local<Object> self = args.Holder();
    Local<Function> super = cons.Get(isolate);

    // => Pet.call(this);
    super->Call(self, 0, NULL);

    // => this.type = "dog";
    self->Set(String::NewFromUtf8(isolate, "type"),
String::NewFromUtf8(isolate, "dog"));

    // 返回自身
    args.GetReturnValue().Set(self);
}

void Init(Local<Object> exports, Local<Object> module)
{
    ...

    Local<FunctionTemplate> dog = FunctionTemplate::New(isolate, Dog);
    dog->Inherit(pet);

    Local<Function> dog_cons = dog->GetFunction();
}

```

```
exports->Set(String::NewFromUtf8(isolate, "Dog"), dog_cons);
}
```

看到了吧，在 Dog 构造函数中，通过 `Local<Function> super = cons.Get(isolate)` 来获取这个 Pet 的构造函数，并且调用它以实现构造函数上的继承。后面的事情就顺理成章了，将 type 设置为 "dog" 以及在 Init 函数中通过 `dog->Inherit` 函数将其继承自 Pet。

关于继承的样例代码就解析到这里了，激动人心的时刻就是去验证它了。

```
$ node-gyp rebuild
...
$ node
> const _ = require("./build/Release/inherit");
undefined
> const pet = new _.Pet();
undefined
> pet
{ name: 'Unknown', type: 'animal' }
> pet.summary();
'Unknown is a/an animal.'
> const dog = new _.Dog();
undefined
> dog
{ name: 'Unknown', type: 'dog' }
> dog.setName("蛋花汤");
undefined
> dog.summary();
'蛋花汤 is a/an dog.'
```

这样的代码执行起来毫无悬念，Dog 真的继承自 Pet 类。

3.6.6 小结

本节讲解了 V8 中的两个重要模板类型——函数模板和对象模板。这两个模板很大意义上是相辅相成的。

除此之外，本节还介绍了这两种模板各自的一些子概念，如函数模板的和对象模板自身的一些 API，以及对象模板的访问器与拦截器、内置字段，还有就是函数模板的继承。

有了这些概念，对于阅读本书的后续章节，以及进行 Node.js 的 C++ 扩展开发都提供了一个较好的前驱知识储备。

3.6.7 参考资料

[1] Embedder's Guide: <https://github.com/v8/v8/wiki/Embedder%27s-Guide>.

[2] How to pass the second parameter of `ObjectTemplate::New` in Chrome V8?: <http://stackoverflow.com/questions/43909594/how-to-pass-the-second-parameter-of-objecttemplateneew-in-google-v8>.

3.7 常用数据类型

前面讲了一些关于 Chrome V8 的概念性内容，包括句柄、作用域和模板等。在大家对 Chrome V8 的思想有了一定认知之后，本节将会对其常用的一些数据类型进行说明，以便让大家对前面的章节能有更明确的认知。

3.7.1 基值（Value）

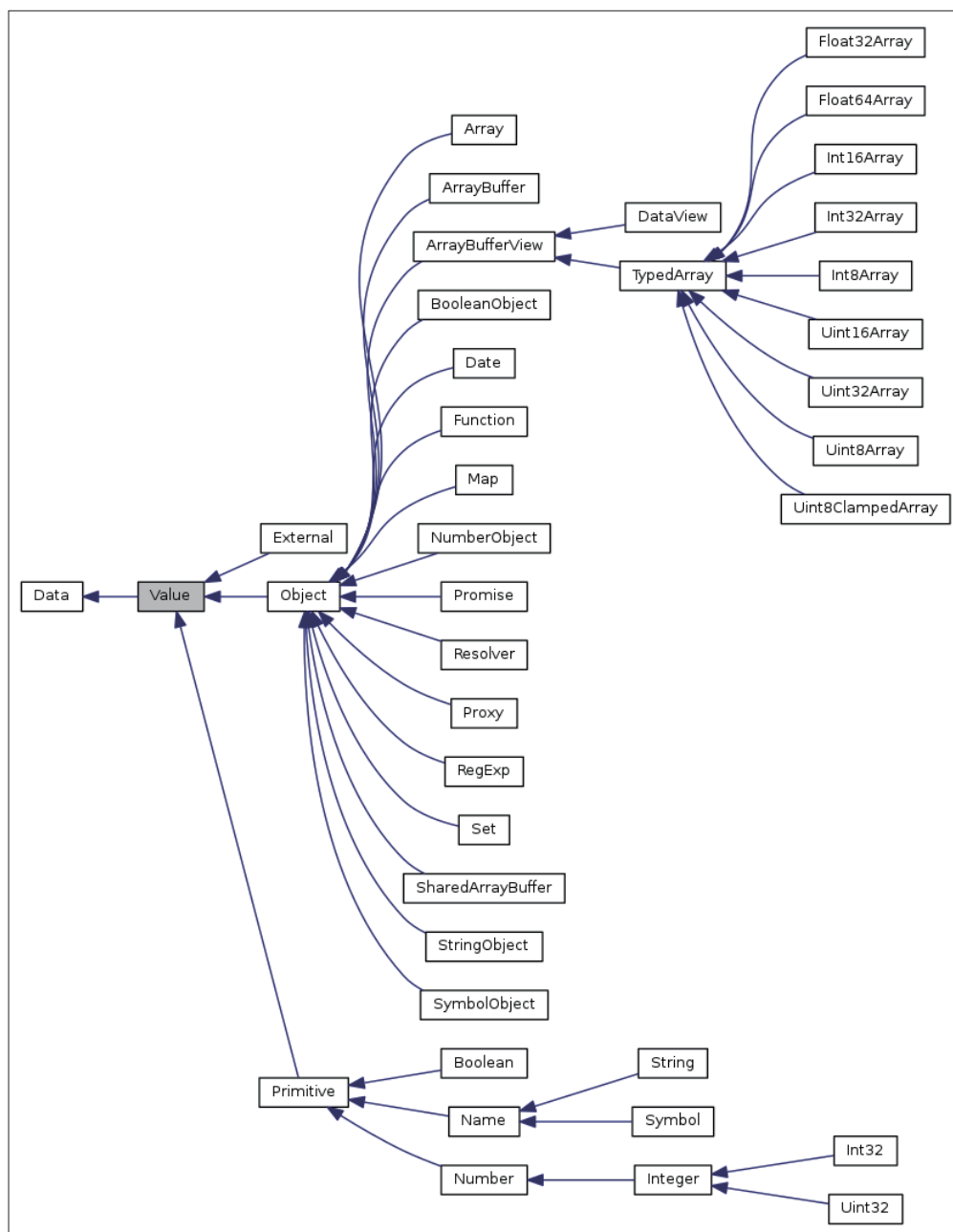
`v8::Value` 是 Chrome V8 在 JavaScript 层面用到的各种数据类型（如 `Number`、`String`、`Function` 等）的一个总的基类，也就是说这些数据类型都是从 `Value` 继承而来的。所以我们经常能从代码中看到 `Value` 类型的本地句柄，也就是 `Local<Value>`。关于 Chrome V8 的基值继承关系图如图 3-3 所示。

由图 3-3 中我们能看到，`Value` 继承自 `Data`，不过 `Data` 不在我们的关心范畴中。而其他类似于笔者先前提到过的 `External` 和 `Object` 均继承自 `Value`，除此之外还有一个元数据类型 `Primitive` 也继承自它。

从这 3 种类型的子类型的继承链我们能找到绝大多数的 JavaScript 类型。

由于 `Value` 是很多 JavaScript 数据类型的父类，因此当你遇到这种数据的句柄时，可以认为它是某一种数据类型的抽象。至于想要知道具体是哪一种数据类型，或者想要将其转换成特定的一种数据类型，就要依靠 `Value` 的各种 API 了。

要知道，我们在声明一个 Node.js 函数的时候所带的参数 `const FunctionCall-backInfo<Value>& args` 中的 `args[]` 操作符返回的是调用这个函数时传入的第几个下标的参数，但由于参数不确定是什么类型的，因此返回的就只是一个 `Local<Value>` 的本地句柄。

图 3-3 基值继承关系图¹

¹ 本图由 Doxygen 生成自 Chrome V8 的文档。

下面列出 Value 两类最主要的 API——Is... 函数和 To... 函数。其他更多的 API 可以参考 V8 相对应版本的文档。

1. Is...

Value 有一类判断该 Value 是哪种数据类型的 API，以 Is 为前缀。如需要判断某个 Value 是否是字符串，就可以调用 Value::IsString() 来判断了。

例如：

```
void Test(const FunctionCallbackInfo<Value>& args)
{
    ...

    // 只是举例，假设传入肯定有一个参数
    // 不考虑容错性
    if(args[0]->IsString())
    {
        return args.GetReturnValue().Set(String::NewFromUtf8(isolate,
"String"));
    }
    else
    if(args[0]->IsNumber())
    {
        return args.GetReturnValue().Set(String::NewFromUtf8(isolate,
"Number"));
    }
    else
    {
        return args.GetReturnValue().Set(String::NewFromUtf8(isolate,
"Other"));
    }
}
```

这个函数的含义就是，判断函数传入的第一个参数是什么类型的，若是 String 或者 Number，就返回相应的类型名字字符串，否则返回 "Other"。所以，Is 函数的作用在这里就可见一斑了。

我们常用的有这些 Is 函数，如表 3-5 所示。

表 3-5 Value 常用的 Is 函数

函数名	说明
IsUndefined	是否为 undefined
IsNull	是否为 null
IsTrue	是否为 true
IsFalse	是否为 false
IsName	是否为一个 Name（即是否为一个字符串或者是符号 Symbol）
IsString	是否为字符串
IsSymbol	是否为符号 ^①
IsFunction	是否为函数
IsArray	是否为数组
IsObject	是否为对象
IsBoolean	是否为布尔类型
IsNumber	是否为数字
IsExternal	是否为 External
IsInt32	是否为 32 位整型
IsUInt32	是否为无符号 32 位整型
IsDate	是否为 Date
IsRegExp	是否为正则表达式
IsPromise	是否为 Promise

① ECMAScript 6 引入了一种新的原始数据类型 Symbol，表示独一无二的值。它是 JavaScript 语言的第 7 种数据类型。

由于篇幅原因，这里仅列举出这些常用的函数。若想了解更多 Is 函数，大家可以参考 Chrome V8 文档¹。

2. To...

To 前缀是 Value 的另一类常用函数，其主要作用就是从这个抽象的 Value 获得另一个类型的句柄，如 ToString() 就是得到一个字符串的待实本地句柄。

在 V8 的漫长迭代中，To 函数也经过了各种变更。在一开始的时候，它的参数是一个 Isolate* 指针，并且返回的是相应的本地句柄。虽然这些函数在目前的 V8 版本中还是处于可用状态的，但是已被宏标记为快被遗弃或者已被遗弃状态，在编译时会发出警告。

就目前来说，比较正确的做法是使用待实本地句柄的版本，传入参数是一个上下文的本地句柄，返回参数是一个待实本地句柄。

1 Node.js v6.x 对应的 V8 关于 Value 的文档地址是 https://v8docs.nodesource.com/node-6.12/dc/d0a/classv8_1_1_value.html。若读者打开该地址，却发现页面不存在，可直接前往 <https://v8docs.nodesource.com/>，并点击“6.x”字样的超链接进入（注意该地址经常换）。

Value 常用的 To 函数参照表 3-6。

表 3-6 Value 常用的 To 函数

原型	说明
<code>MaybeLocal<Boolean> ToBoolean(Local<Context> context) const</code>	返回该 Value 对应的布尔型待实本地句柄
<code>MaybeLocal<Number> ToNumber(Local<Context> context) const</code>	返回该 Value 对应的数字待实本地句柄
<code>MaybeLocal<String> ToString(Local<Context> context) const</code>	返回该 Value 对应的字符串待实本地句柄
<code>MaybeLocal<Object> ToObject(Local<Context> context) const</code>	返回该 Value 对应的对象待实本地句柄
<code>MaybeLocal<Integer> ToInteger(Local<Context> context) const</code>	返回该 Value 对应的整型待实本地句柄
<code>MaybeLocal<Uint32> ToUint32(Local<Context> context) const</code>	返回该 Value 对应的 32 位无符号整型待实本地句柄
<code>MaybeLocal<Int32> ToInt32(Local<Context> context) const</code>	返回该 Value 对应的 32 位整型待实本地句柄

在正常使用中，可以参考这样一段样例代码：

```
void Plus(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();

    // 通过 Isolate 获取当前的上下文
    Local<Context> context = isolate->GetCurrentContext();

    // 将第一个参数和第二个参数分别转换为 Number 型本地句柄
    MaybeLocal<Number> ma = args[0]->ToNumber(context);
    MaybeLocal<Number> mb = args[1]->ToNumber(context);
    Local<Number> a = ma.ToLocalChecked();
    Local<Number> b = mb.ToLocalChecked();

    // a + b
    Local<Number> ret = Number::New(isolate, a->Value() + b->Value());
    args.GetReturnValue().Set(isolate, ret);
}
```

这段没有容错处理的代码流程很简单，就是获取第一个参数和第二个参数并相加，然后返回结果。所以其实最核心的就是通过 `isolate->GetCurrentContext()` 来获取当前上下文，然后再通过这个上下文将 Value 类型的本地句柄转换为 Number 类型的本地句柄。

3.7.2 字符串 (String)

字符串在 Node.js 甚至是 JavaScript 中是一种极其常用的数据类型。通过前面的图 3-3 我们就知道，它继承自 Primitive 的 Name 类型。

字符串有一些自带的 API 和一些在 C++ 中使用的常用技巧。

这里列出几个简易的函数：

- `String::Length()`：返回字符串的长度。
- `String::Utf8Length()`：返回字符串的 UTF8 长度，把 UTF8 拆成字节。
- `String::IsOneByte()`：该字符串是否只包含一个字节数据。

其他函数还是按老规矩来，可以参考 V8 的相应版本文档。

不过就上面提及的一些方法来说，还不足以做到让开发者在 Node.js C++ 扩展开发时方便地与 C++ 底层数据交互。其实除此之外，还有一些函数和数据类型是能让 String 与 `char*` 等数据类型直接进行转换的。

1. 从 `char*` 转 String

String 数据类型有多个静态函数可以让开发者从一个 `char*` 指针建立起一个 V8 字符串数据。不过这里笔者介绍一个最常用的函数——String 的静态函数 `NewFromUtf8`。

这个函数大家应该在前面的一些章节中经常看到，顾名思义，其就是从一个 UTF8 数据中新建一个 String 数据。

函数原型如下所示：

```
static MaybeLocal<String> String::NewFromUtf8(Isolate* isolate, const char*
data, v8::NewStringType type, int length = -1);
```

与 Value 类似，`NewFromUtf8` 一样有快被遗弃的非 `MaybeLocal` 版本，所以读者在本书中见到使用 `NewFromUtf8` 时并未执行 `ToLocalChecked` 也不用觉得奇怪——至少它们在当下版本中是可以用的。

这个函数有 4 个参数，但通常我们只需要用前面两个就够了。对于第 3 个参数 `v8::NewStringType`，我们通常传入 `kNormalString` 即可。

跟着笔者再来温习一下 `NewFromUtf8` 的用法。

```
MaybeLocal<String> ms = String::NewFromUtf8(isolate, "这是一个字符串。",
kNormalString);
Local<String> s = ms.ToLocalChecked();
```

2. Utf8Value——根据 String 获取 char*

一个 String 类型不方便在 C++ 底层做一些计算，如字符串查找、使用、拼接等，所以我们需要将其转换成自己能直接使用的 char* 来。

首先我们用到的是一个 v8::Utf8Value 类型。这个数据类型用于将一个对象转换成 UTF8 编码的字符数组。

具体用法是在其构造函数时传入我们要转换的字符串。

```
// 假设 s 就是刚才那个 " 这是一个字符串 "
Utf8Value v(s);
```

在得到它之后，使用 * 这个被重载的操作符获取 char* 或者 const char*。我们来看一下这个 Utf8Value 的原型：

```
class Utf8Value {
public:
    explicit Utf8Value(Local<v8::Value> obj);
    ~Utf8Value();
    char* operator*() { return str_; }
    const char* operator*() const { return str_; }
    int length() const { return length_; }
private:
    char* str_;
    int length_;

    // 禁止复制和赋值
    Utf8Value(const Utf8Value&);
    void operator=(const Utf8Value&);
};
```

从类的声明来看，* 这个操作符就是用来返回字符串用的。

也就是说，从 String 得到 char* 的一个通用流程可以这么写：

```
// 假设 str 是一个字符串的本地句柄
Utf8Value some_value(str);
char* ret = *some_value;

printf("得到的字符串: %s\n", ret);
```

最后值得一提的是，通过 Utf8Value 转换字符串时结果可能会失败，这个时候 Utf8Value 的 length() 函数将会返回 0，并且 * 操作符会返回一个空指针（也就是 NULL）。

3.7.3 数值类型

数值类型在 V8 中代表的意义很宽泛。首先，在 ECMAScript 的定义中，`Number` 是一个 IEEE 754 标准的 64 位双精度浮点数类型¹。但是在 ECMAScript 标准中还定义了，当数值需要各种转换的时候（如 `ToInteger` 或者 `ToInt32` 等），有一些中间类型的数据也属于数值类型²。

由于这些中间数值类型是从 `Number` 继承出来的，因此它也属于 V8 的数值类型。

这些类型如下。

- `Integer`：继承自 `Number`。
- `Int32`：继承自 `Integer`。
- `Uint32`：继承自 `Integer`。

关于数值类型的用法很简单，常用的无非是静态函数 `New()` 以及成员函数 `Value()`。

```
// `Value()` 函数声明，返回一个 `double` 数值
double Number::Value() const;

// `New()` 函数声明，返回一个数值的本地句柄
static Local<Number> New(Isolate* isolate, double value);
```

相应地，`Integer` 以及其他几个数值类型也有其相应的 `New()` 函数和 `Value()` 函数。

不过值得注意的是，`Integer::Value()` 的返回值是 `int64_t` 类型的数据，但是在 `New()` 的时候传的却需要是 `int32_t` 或者 `uint32_t`。

3.7.4 布尔类型（Boolean）

布尔类型非常简单，常用的 API 与数值类型差不多，无非是 `New()` 和 `Value()` 两个，不同的是它们的参数或者返回值是一个 `bool` 类型罢了。

3.7.5 对象（Object）

对象是我们在 C++ 扩展中非常常用的一个数据类型。从 `Object` 出发，衍生了各种其他非元类型的数据类型，如数组、函数等。

¹ 该定义在 ECMAScript 5.1 中就有体现，在其 4.3.19 节中。

² 在 ECMAScript 5.1 的第 9 节中就有体现，当一些数据需要转换的时候，就有可能用到 `ToInteger` 等函数将其转换为中间数值类型 `Integer` 等。如 `Array.prototype.slice` 中的 `start` 和 `end` 参数，就会被转换为 `Integer` 这个中间类型使用。

对象的一些内容（如拦截器等）在 3.6.2 节中曾有所提及。

本节将介绍它的几个主要 API。如果在本书的其他章节中出现了本节未提及的 API，请不要害怕，可以尝试参考一下 V8 的相应版本文档——毕竟本书的重点不是介绍 API。

1. New

与其他类型一样，对象也有 `New()` 静态函数。

```
Local<Object> obj = Object::New(isolate);
```

这里的代码表示新建了一个全新的空数据对象。

2. Set、Get 和 Delete 等

`Set` 和 `Get` 是两个最主要的函数。即使是在 JavaScript 层面，一个对象最常用的也是获取和设置某个值。

与前面的一些类型相似，在迭代中，抛弃了一些现在还能使用但却不是特别安全的 API。例如：

```
bool Object::Set(Local<Value> key, Local<Value> value);
```

上面这个声明就是旧版的（现在还能使用的）设置某个属性的 API，读者在前面章节的一些样例代码中经常能看到。

不过我们也应该向安全的 API 靠近：

```
// 安全的 Set 和 Get 的声明
Maybe<bool> Object::Set(Local<Context> context, Local<Value> key,
Local<Value> value);
Maybe<bool> Object::Delete(Local<Context> context, Local<Value> key);
MaybeLocal<Value> Object::Get(Local<Context> context, Local<Value> key);

// 数组下标版本的声明
Maybe<bool> Object::Set(Local<Context> context, uint32_t index, Local<Value>
value);
Maybe<bool> Object::Delete(Local<Context> context, uint32_t index);
MaybeLocal<Value> Object::Get(Local<Context> context, uint32_t index);
```

其中，`Maybe<bool>` 这种类型是用来确认结果用的。

Maybe¹

Maybe 是一个简单的用于表现一个对象是否具值的数据类型。

当一个 API 返回一个 `Maybe<>` 时,就说明它可能是一个布尔值,也可能是一个因为异常而得到的无值结果。

这种 `Maybe<>` 的数据类型有几个常用的函数。

- `bool Maybe<T>::IsNothing() const`: 是否具值。
- `bool Maybe<T>::IsJust() const`: 与上面的这个函数结果相反。
- `T Maybe<T>::FromJust() const`: 返回它本体的值,如果不具值则直接崩溃。
- `T Maybe<T>::FromMaybe(const Maybe& default_value) const`: 返回它本体的值,如果不具值则返回 `default_value`。

不过基于 *Let it crash*² 的原则,通常我们在校验结果的时候均直接通过 `FromJust()` 来触发异常,从而使进程崩溃。事实上 Node.js 的源码中就是这么做的。如:

```
ttls->Set(context, i, value).FromJust();
```

其代表的意思就是当 `ttls` 设置值之后,返回的 `Maybe<>` 如果不具值就直接崩溃。

当然,如果你明确知道当一个 `Maybe<>` 返回了空值的时候需要做哪些异常容错处理,也可以不让它崩溃,而选择使用另两个 API。

读者现在依然能用老的 API 进行开发,不过如果使用新版的安全 API,应该跟下面这样差不多:

```
// 新版 API
MaybeLocal<String> mkey = String::NewFromUtf8(isolate, "value",
kNormalString);
Local<String> key = mkey.ToLocalChecked();
Local<Number> value = Number::New(isolate, 2333);
obj->Set(context, key, value).FromJust();
Local<Value> val = obj->Get(context, key).ToLocalChecked();
Local<Number> num = val->ToNumber(context).ToLocalChecked();

// 旧版 API
Local<String> key = String::NewFromUtf8(isolate, "value");
Local<Number> value = Number::New(isolate, 2333);
```

¹ V8 的 `Maybe<>` 思想来自 Haskell 的 `Data.Maybe`, 可以参考 <https://hackage.haskell.org/package/base-4.9.1.0/docs/Data-Maybe.html>。

² 即任其崩溃原则,在 Erlang 中比较普及,但这并不是其专利。别去捕获你不知道该怎么处理的异常,由它去吧;否则当程序继续运行在一个不可预知的已经处于异常状态的环境下,可能造成的损失会更大。

```
obj->Set(key, value);
Local<Value> val = obj->Get(key);
Local<Nubmer> num = val->ToNumber();
```

3. 内置字段的相关内容

笔者在 3.6.4 节中提到过对象模板的内置字段，也顺带地介绍了一下对象数据类型的内置字段使用方法。

这里先列出关于内置字段的一些 API：

- InternalFieldCount
- GetInternalField
- SetInternalField
- GetAlignedPointerFromInternalField
- SetAlignedPointerInInternalField

读者可能还记得，前面 3 个 API 之前提到过并且使用过了，所以这里就不再赘述。下面介绍一下 `GetAlignedPointerFromInternalField` 和 `SetAlignedPointerInInternalField` 两个函数。

我们记得之前的两个函数传入的内置字段必须是由 `External` 包裹而成的，但是每次这么做都会让人感觉有些小麻烦。现在介绍的这两个函数就是为了简化这种麻烦。

其中 `SetAlignedPointerInInternalField` 函数的作用就是在内置字段中，以两字节对齐的方式将其写入相应的位置，而不是将一个 `External` 数据类型写入。这样确实是方便了，不过要注意的一点是，通过 `SetAlignedPointerInInternalField` 写入的两字节对齐内置字段必须使用 `GetAlignedPointerFromInternalField` 将其取出，否则直接使用 `GetInternalField` 的话就会失败。

下面看看这两个函数的定义吧。

```
void* Object::GetAlignedPointerFromInternalField(int index);
void Object::SetAlignedPointerInInternalField(int index, void* value);
```

例如，我们新的设置内置字段姿势可以是这样的：

```
Person* person = ...;

Local<Object> handle = Object::New(isolate);
handle->SetAlignedPointerInInternalField(0, person);
```

```
Person* gotten = reinterpret_cast<Person*>(handle-> GetAlignedPointerFromInternalField(0));
```

如你所见，与先前介绍的两个函数相比，这两个函数使用起来更为方便。不过即使是使用这种姿势，对于使用弱持久句柄来管理其生命周期的这一点，也需要跟以前一样去写。

3.7.6 函数（Function）

别忘了函数也是对象的一种，所以说 V8 中的 Function 也是继承自 Object 的。对于外界传进来的 Value 类型的函数，读者能通过 3.3.1 节介绍过的本地句柄的静态函数 Local<T>::Cast 来将其转换为函数类型；如果是对于自身写的 C++ 函数，我们通常会通过函数模板的方式将其实例化出来，这一点在 3.6.1 节中已经介绍过了。

而对于一个已经是函数类型的数据来说，读者能对它做以下一些事情。

- Call()：调用这个函数。
- NewInstance()：相当于通过 new 的方式调用这个函数，以得到类的实例。
- SetName() / GetName()：设置、获取函数名。
- 其他函数请参考 API 文档。

本节着重介绍的是如何调用一个函数的数据类型。

1. 函数调用（Call）

读者先看看 Call 的一个函数声明吧：

```
// 新版
MaybeLocal<Value> Function::Call(Local<Context> context, Local<Value> recv,
int argc, Local<Value> argv[]);

// 旧版
Local<Value> Function::Call(Local<Value> recv, int argc, Local<Value>
argv[]);
```

针对新版的 Call 函数，它的各参数含义如下：

- context：上下文。
- recv：相当于被调用函数内部的 this，参照 Function.prototype.apply¹。
- argc：这次函数调用的参数个数。
- argv：与参数个数对应的参数数组，以本地 Value 句柄的形式出现。

¹ Function.prototype.apply 有两个参数：第一个参数指定 this，第二个参数是传入函数中调用的参数数组。

如果大家觉得很模糊的话，可以跟着笔者的脚步来实现一个类似于 `Array.prototype.map` 的函数来看看。这个时候就需要打开“11. array prototype map”中的 `map.cc` 来看看了。

对于 `Init` 函数这里就不做过多的解释了，我们主要把目光放到 `Map` 函数中。

```
void Map(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();
    Local<Context> context = isolate->GetCurrentContext();

    Local<Array> array = args[0].As<Array>();
    Local<Function> func = args[1].As<Function>();

    Local<Array> ret = Array::New(isolate, array->Length());

    Local<Value> null = v8::Null(isolate);
    Local<Value> a[3] = { Local<Object>(), null, array };
    for(uint32_t i = 0; i < array->Length(); i++)
    {
        a[0] = array->Get(i);
        a[1] = Int32::New(isolate, i);

        MaybeLocal<Value> v = func->Call(context, null, 3, a);
        ret->Set(i, v.ToLocalChecked());
    }

    args.GetReturnValue().Set(ret);
}
```

在函数中，首先获取第一个参数和第二个参数作为数组和回调函数，分别为 `array` 与 `func`。接下来以一个 `for` 循环打开新世界的大门，逐个遍历 `array` 的元素，并将其与它的下标和 `array` 本身一起传入回调函数 `func` 中，得到结果 `MaybeLocal<Value> v`；将结果 `v` 设置到事先声明好的 `ret` 数组的相应下标中。最后返回这个结果 `ret`。

`Map` 函数原型参照了 JavaScript 的 `Array.prototype.map`¹，其回调函数也有 3 个参数，分别为元素本体、元素下标与整个数组。

写好之后自然要好好地运行一遍了。不过同样地，由于篇幅所限，这里的代码也没做容错处理，而是直接执行 `ToLocalChecked` 了。

¹ 不熟悉 `Array.prototype.map` 的读者可以参阅 JavaScript 文档的相应章节，可访问 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map。

```
$ node-gyp rebuild
...
$ node
> const map = require("../build/Release/map");
undefined
> map([ " 芙兰朵露 ", " 蛋花汤 ", " 南瓜饼 " ], function() { return arguments; });
[ { '0': ' 芙兰朵露 ', '1': 0, '2': [ ' 芙兰朵露 ', ' 蛋花汤 ', ' 南瓜饼 ' ] },
  { '0': ' 蛋花汤 ', '1': 1, '2': [ ' 芙兰朵露 ', ' 蛋花汤 ', ' 南瓜饼 ' ] },
  { '0': ' 南瓜饼 ', '1': 2, '2': [ ' 芙兰朵露 ', ' 蛋花汤 ', ' 南瓜饼 ' ] } ]
```

2. 构造函数的实例化 (NewInstance)

至于 NewInstance 函数，它的声明是这样的：

```
MaybeLocal<Object> Function::NewInstance(Local<Context> context, int argc,
Local<Value> argv[]);
```

通过这个函数传入参数的个数和具体的参数数组，读者能得到类似于通过 new 调用某个构造函数的实例化结果。

3.7.7 数组 (Array)

数组也继承自对象，通常在转换的时候由句柄的 As 函数来完成。这一点在 3.7.6 节的 Map 代码中也有体现。

```
Local<Array> array = args[0].As<Array>();
```

除此之外，这里再介绍一下 Array 的几个常用 API。

1. New

与对象不同的是，数组的 New 函数还可以多带一个参数，代表该数组的长度。详细信息参见“11. array prototype map”中的相应代码。

2. Set 与 Get

这里的 API 在 3.7.5 节中也已介绍，主要使用下标的形式来设置和获取。详细信息参见“11. array prototype map”。

3. Length

获取数组的长度。其与下面的代码等效：

```
MaybeLocal<Value> v = array->Get(
    context,
    String::NewFromUtf8(isolate, "length"));
Local<Uint32> len = v.ToLocalChecked().As<Uint32>();
uint32_t length = len->Value();
```

3.7.8 JSON 解析器

Chrome V8 的 JSON 解析器也充满了黑科技，所以读者在需要解析 JSON 字符串时通常不需要自己去实现一套或者另找第三方库，而是直接使用 V8 自带的 JSON 解析器即可。

JSON 在 V8 中是一个类，其有两个静态函数。

- `static Local<Value> JSON::Parse(Local<String> json_string)`
- `static MaybeLocal<Value> JSON::Parse(Isolate *isolate, Local<String> json_string)`

相信大家又猜到了：这两个函数都可用，只不过前者属于快被遗弃的版本。

3.7.9 函数回调信息（Function Callback Info）

3.6.1 节提到过，要实例化一个 `Function` 对象，首先要用 `FunctionTemplate` 对一个特定函数进行封装。而那个所谓的特定函数的参数是 `FunctionCallbackInfo` 类型的，这个类型就叫 V8 的函数回调信息，其中包含了一个 JavaScript 函数调用所需的各种信息。

并且这个回调信息是一个模板类，模板类型代表了传入参数的类型。通常读者使用抽象数据类型 `Value` 即可。

假设我们有一个 JavaScript 函数声明如下：

```
void Func(FunctionCallbackInfo<Value>& args)
{
}
```

是不是很眼熟？虽然前面大家都轻车熟路了，不过这里还是再系统性地介绍一下这个类型吧。

在函数内部使用 `args` 的时候，就有这些函数可以使用。

- `Length()`：返回传入的参数个数。
- `[]` 重载：使用 `args[i]` 返回第 `i` 个参数，并且是 `Value` 类型的。
- `This()`：得到函数的 `this`。

- `Holder()`: 得到函数调用时的 `this`, 使用 `Function.prototype.bind` 等可改变。
- `IsConstructCall()`: 该次调用是否是构造函数调用, 即是否使用 `new` 前缀进行调用。
- `GetIsolate()`: 获取当前的 `Isolate`。
- `GetReturnValue()`: 获取存储返回值的对象, 可对其设置函数返回值。

3.7.10 函数返回值 (Return Value)

函数返回值类型与函数回调信息紧密相连。通过 `GetReturnValue()` 得到的对象就是一个 `ReturnValue` 类型的数据体。

读者能通过各种 `Set` 函数来设置其返回值。

- `Set()`: 可传入一个任意数据类型的句柄。无论是本地句柄还是持久句柄都可以; 除此之外, 还能便捷地传入 `bool`、`double`、`int32_t` 和 `uint32_t` 这类基本类型, 在返回的时候 V8 会自行转换。
- `SetNull()`: 返回一个 `null`。
- `SetUndefined()`: 返回一个 `undefined`。
- `SetEmptyString()`: 返回一个空字符串。

3.7.11 隔离实例 (Isolate)

隔离实例在 3.2.2 节中曾介绍过, 本节介绍一下它的 3 个常用 API。

1. GetCurrent 静态函数

`Isolate::GetCurrent()` 是一个静态函数, 用于返回当前 Node.js 所使用的 `Isolate` 指针。而通常在 Node.js 的函数中, 也可以通过 `args.GetIsolate()` 来获取隔离实例, 在其他无法通过这种方式来获取隔离实例的地方, 若需要用到它, 则推荐使用参数形式将其传入。

不过如果当你真的处于一个没法获取 `Isolate` 指针的时刻, 就可以使用该 `Isolate::GetCurrent()` 函数来获取当前的隔离实例了。

2. GetCurrentContext

这个函数用于获取当前的上下文本地句柄。上下文本地句柄在当前版本的 V8 中作为很多安全函数所必要的参数之一存在, 所以这个函数也是比较常用的。

```
Isolate* isolate = args.GetIsolate();
Local<Context> context = isolate->GetCurrentContext();
```

3. ThrowException

下面先给出函数原型：

```
Local<Value> Isolate::ThrowException(Local<Value> exception);
```

这相当于 JavaScript 代码中的 `throw` 操作，抛出一个指定的异常。由于异常不一定是 `Error` 类型的，因此这个函数的参数也是抽象的 `Local<Value>` 类型。

3.7.12 小结

本节讲述了 Chrome V8 在 JavaScript 层面的一些基本数据类型。常用的类型如下：

- Value
- String
- Number
- Integer
- Int32
- Uint32
- Boolean
- Object
- Function
- Array
- FunctionCallbackInfo
- ReturnValue

而所有这些数据类型的 API 都能在其文档中被找到。出于时效性和篇幅考虑，本书中只介绍了这些常用的数据类型中最常用的一些 API。可能大家在阅读本书的时候，V8 的迭代又发生了变化。

不过千变万化，都不离其宗，所有的数据类型和其 API 都能通过阅读由 Doxygen¹ 生成的相应 V8 版本文档来获得。具体的 Chrome V8 文档可以访问 <https://v8docs.nodesource.com/> 获取，其中列出了若干版本的 Node.js，大家可以根据自己的需求来选择相应的版本。

有了这些基础之后，对于复习前面的一些章节，以及阅读后面的章节均提供了非常好的知识储备。

¹ Doxygen 是一种开源跨平台的，以类似于 JavaDoc 风格描述的文档系统，其完全支持 C、C++、Java、Objective-C 和 IDL 语言，部分支持 PHP、C#。可参考 <http://www.stack.nl/~dimitri/doxygen/>。

3.7.13 参考资料

- [1] ECMAScript® Language Specification: <https://www.ecma-international.org/ecma-262/5.1/#sec-4.3.19>.
- [2] 应该如何理解 Erlang 的“任其崩溃”思想? : <https://www.zhihu.com/question/21325941>.
- [3] JavaScript Reference: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>.

3.8 异常机制

在 Chrome V8 中，有一套关于 JavaScript 中的异常处理办法，其相当于 JavaScript 中的 `try-catch`，而非 C++ 层面的异常处理。

3.8.1 try-catch

`TryCatch` 是 V8 中一个捕获 JavaScript 异常的类，管辖的是其生命周期中的 V8 层面异常。即，一个 `TryCatch` 对象相当于在其声明周期内创建了一个 JavaScript 形式的 `try-catch` 代码块。

在 `TryCatch` 的对象中，有几个重要的 API。

- 构造函数：传入的是 `Isolate*` 指针。
- `bool HasCaught()`：是否有错误被该 `TryCatch` 域捕获。
- `Local<Value> Exception()`：返回一个 `Exception` 对象，代表捕获的错误实体。
- `Local<Value> ReThrow()`：重新将其捕获的错误通过 `throw` 抛出去。

其余的 API 可以在 V8 文档中寻找。

在介绍完 `TryCatch` 类之后，大家可以尝试写一段关于 JavaScript 异常处理的代码：

```
function get(obj, key) {  
    try {  
        return obj[key];  
    } catch(e) {  
        throw e;  
    }  
}
```

乍一看，里面的 try-catch 加上 throw 像是多此一举，但是如果你将这段代码迁移成 C++ 代码的话，就会发现还是有必要的。

请移步至“12. try catch”观看代码。

```
void Get(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();
    Local<Context> context = isolate->GetCurrentContext();
    TryCatch trycatch(isolate);

    MaybeLocal<Object> maybe_obj = args[0]->ToObject(context);
    MaybeLocal<String> maybe_key = args[1]->ToString(context);

    if(maybe_obj.IsEmpty() || maybe_key.IsEmpty())
    {
        if(trycatch.HasCaught())
            trycatch.ReThrow();
        else
            args.GetReturnValue().Set(v8::Undefined(isolate));

        return;
    }

    Local<Object> obj = maybe_obj.ToLocalChecked();
    Local<String> key = maybe_key.ToLocalChecked();

    Local<Value> ret = obj->Get(context, key).ToLocalChecked();
    args.GetReturnValue().Set(ret);
}
```

该函数的意义与刚才那段 JavaScript 代码大体一致，通过两个 MaybeLocal 句柄获取两个参数并转换为对象和字符串。

当两个 MaybeLocal 句柄的任意一个为空的时候，就说明 V8 在做类型转换的时候出了问题。如当 args[0] 本体实际上是一个 null 或者 undefined 的时候，生成的 maybe_obj 就会是一个空句柄。这个时候就需要判断一下是否在转换的时候有异常被 TryCatch 捕获到了，即通过 trycatch.HasCaught() 判断。若在转换过程中的确有异常产生，那么笔者就将异常抛给上级作用域；否则，由于不知道是什么异常，笔者就让这个 JavaScript 函数返回 undefined。剩下的就是正常情况了，这时候只需要返回 obj->Get(context, key) 即可。

在测试时，也可以尝试着将中间那段异常处理的 if 分支注释掉，即完全不要 if(maybe_obj.IsEmpty() ...) 一整段处理。

下面分两份代码测试结果吧。

1. 没有异常处理

```
$ node-gyp rebuild
...
$ node
> const get = require("../build/Release/try_catch");
undefined
> get({ a: 1 }, "a");
1
> get(null, "a");
FATAL ERROR: v8::ToLocalChecked Empty MaybeLocal.
 1: node::Abort() [/Users/USER/.nvm/versions/node/v6.9.4/bin/node]
 2: node::FatalException(v8::Isolate*, v8::Local<v8::Value>,
v8::Local<v8::Message>) [/Users/USER/.nvm/versions/node/v6.9.4/bin/node]
 3: v8::V8::ToLocalEmpty() [/Users/USER/.nvm/versions/node/v6.9.4/bin/node]
 4: __try_catch__::Get(v8::FunctionCallbackInfo<v8::Value> const&) [/Users/
USER/nyaa-nodejs-demo/12. try_catch/build/Release/try_catch.node]
 5: v8::internal::FunctionCallbackArguments::Call(void (*)(v8::FunctionCallb
ackInfo<v8::Value> const&)) [/Users/USER/.nvm/versions/node/v6.9.4/bin/node]
 6: v8::internal::MaybeHandle<v8::internal::Object> v8::internal::(anonymous
namespace)::HandleApiCallHelper<false>(v8::internal::Isolate*,
v8::internal::(anonymous namespace)::BuiltinArguments<(v8::internal::Builtin
ExtraArguments)1>) [/Users/USER/.nvm/versions/node/v6.9.4/bin/node]
 7: v8::internal::Builtin_HandleApiCall(int, v8::internal::Object**,
v8::internal::Isolate*) [/Users/USER/.nvm/versions/node/v6.9.4/bin/node]
 8: 0x6021c1092a7
[1] 12805 abort node --harmony
```

从运行结果中可以看到，`get({ a: 1 }, "a")` 是正常的。但是后者从 `null` 中获取 "a" 的时候，整个 Node.js 进程就直接崩溃退出了。

因为笔者在 `MaybeLocal<Object> maybe_obj = args[0]->ToObject(context)` 这段代码执行的时候得到了一个空待实本地句柄；然后并没有在意是什么异常导致了空待实句柄的产生，而是直接调用 `Local<Object> obj = maybe_obj.ToLocalChecked()` 将空待实本地句柄转换成本地句柄了。第二步的直接转换导致了 Chrome V8 层面的崩溃——对于这种崩溃，哪怕你在 JavaScript 调用这个函数时加上 `try-catch` 也无济于事。

2. 有异常处理

```
$ node-gyp rebuild
...
```



```

$ node
> const get = require("../build/Release/try_catch");
undefined
> get({ a: 1 }, "a");
1
> get(null, "a");
TypeError: Cannot convert undefined or null to object
    at repl:1:12
    at sigintHandlersWrap (vm.js:96:12)
    at REPLServer.defaultEval (repl.js:313:29)
    at bound (domain.js:280:14)
    at REPLServer.<anonymous> (repl.js:513:10)
    at emitOne (events.js:101:20)
    at REPLServer.emit (events.js:188:7)
    at REPLServer.Interface._onLine (readline.js:239:10)
    at REPLServer.Interface._line (readline.js:585:8)
    at REPLServer.Interface._ttyWrite (readline.js:862:14)

```

当上面的那段代码有异常处理时，异常就是 JavaScript 层面抛出的，这时只需要在调用这个函数的时候加上 try-catch 块就能正常运行了。正如代码中写的一样，当笔者探测到待实本地句柄是空的时候就觉得是有异常了，这时避免直接将待实本地句柄转换为本地句柄；否则会发生 V8 层面的崩溃，这样就万劫不复了。代替直接转换，我们将得到的异常再以 JavaScript 代码能够识别的异常形式抛出去，交给上层处理即可。

当然，读者还能再改一下代码，连异常也不继续抛出，直接返回 undefined 也是可以的，就像这样：

```

if(maybe_obj.IsEmpty() || maybe_key.IsEmpty())
{
    args.GetReturnValue().Set(v8::Undefined(isolate));
    return;
}

```

不过，至于如何选择还要看开发者当时的具体业务场景。

3.8.2 抛出异常

除了在自己代码中去捕获异常进行处理，读者也能在自己的 C++ 层面代码对 JavaScript 抛出一个异常供其捕获用。一个最简单的例子就是，外部 JavaScript 调用 C++ 扩展函数时，参数类型或者数量不吻合就可以往外抛出一个异常了。

一种抛出异常的方法在 3.8.1 节中介绍过，就是利用 `TryCatch::ReThrow()` 函数，它可以将一个已捕获的异常继续通过 `throw` 抛出去。

另一种方法就是通过 `Isolate` 隔离实例来抛出异常。

`Isolate` 有一个成员函数叫 `ThrowException`，`ThrowException` 函数接收一个参数，代表要抛出的错误内容。由于 JavaScript 的 `throw` 可以抛出任意类型的异常，因此在该函数中的错误内容是一个 `Value` 的本地句柄。不过在实际开发中，还是推荐抛出一个异常的错误对象（即 `Exception`）。

下面打开“12. try catch”中的 `try_catch.cc` 来看一看代码。

```
void Div(const FunctionCallbackInfo<Value>& args)
{
    // 篇幅所限，使用旧版 API
    Isolate* isolate = args.GetIsolate();
    Local<Int32> a = args[0]->ToInt32();
    Local<Int32> b = args[1]->ToInt32();

    int32_t av = a->Value();
    int32_t bv = b->Value();

    if(bv == 0)
    {
        isolate->ThrowException(Exception::Error(String::NewFromUtf8(
(isolate, "除数不能为 0。"))));
        return;
    }

    args.GetReturnValue().Set(av / bv);
}
```

这是一个整除的函数。在 JavaScript 中，如果除数为 0，那么 `/` 操作符返回的结果会是 `Infinity`、`-Infinity` 或者 `NaN` 这类非常规数值类型。如果想要在除数为 0 时返回的不是这些值，而是直接抛出异常，那就是这段样例代码所实现的逻辑了。

首先获取函数的两个参数 `av` 和 `bv`，其中 `bv` 是除数。然后判断 `bv` 是否为 0。若 `bv` 不为 0 则返回它们整除的结果；若为 0，则通过 `isolate` 抛出一个异常，内容是“除数不能为 0。”。

注意：当读者要抛出异常的时候，就不应该再通过 `args.GetReturnValue().Set()` 来设置返回值的内容了，而是应该直接退出函数。

3.8.3 异常生成类 (Exception)

介绍了异常捕获和抛出后，接下来介绍错误对象。异常生成类在 Chrome V8 层面叫 **Exception** 类，通过这个类中的一些静态函数能返回错误对象，这些错误对象对应到 JavaScript 层面就是 **Error** 对象。

我们在 3.8.2 节中看到，通过 `isolate` 抛出的异常是一个调用 `Exception::Error` 生成的错误对象。实际上，`Exception` 不止这一个静态函数。

- `static Local<Value> Exception::RangeError(Local<String> message):` 生成一个 `RangeError` 错误对象。
- `static Local<Value> Exception::ReferenceError(Local<String> message):` 生成一个 `ReferenceError` 错误对象。
- `static Local<Value> Exception::SyntaxError(Local<String> message):` 生成一个 `SyntaxError` 错误对象。
- `static Local<Value> Exception::Error(Local<String> message):` 生成一个普通的 `Error` 错误对象。

看了这 4 个函数之后，可能有人会有疑问，为什么会有这么多不同的错误类型。在我们进行 Node.js 的 C++ 扩展开发的时候，其实任何时候使用普通 `Error` 都可以，使用其他错误类型主要是为了一个语义的严谨性。如这个错误表示的是数据范围错误，就应该用 `RangeError`。当然即使这时候使用 `Error` 的话，虽然语义逻辑上有点不严谨，但是代码逻辑却是可以正常运行的。

3.8.4 小结

本节介绍了如何在 Chrome V8 层面对异常进行处理。

其中 `TryCatch` 对象对应 JavaScript 代码中的 `try-catch` 代码块，`Isolate::ThrowException` 对应 JavaScript 的 `throw` 操作。

最后，`Exception::Error` 函数对应 `new Error` 操作，并且这个静态类能生成的不只是普通的 `Error` 对象，其还能生成诸如 `SyntaxError` 等错误对象。

3.8.5 参考资料

[1] Embedder's Guide: <https://github.com/v8/v8/wiki/Embedder%27s-Guide>.

4

C++ 扩展实战初探

本章将开始讲解 Node.js 的 C++ 扩展编写方式、目录结构等内容。

在读者积累了第 3 章的 Chrome V8 基础知识后，本章从 `binding.gyp`、代码结构、Node.js C++ 帮助类等方面进行讲解，以使读者对 C++ 扩展的编写有一个更系统的认知。

4.1 `binding.gyp`

在阅读本节内容之前，建议读者再回顾一下 1.4.2 节的内容，因为 `binding.gyp` 文件就是一个谷歌的编译工具 `gyp` 文件，且是通过 `node-gyp` 命令对源码进行编译的。本节会详细介绍一下这个定义 C++ 扩展编译行为的文件 `binding.gyp`。

在每一个 Node.js 包中，包括自己的非包类项目，如果需要编译 C++ 代码，最通用的做法就是把你的代码放到某一个目录中，然后通过 `binding.gyp` 文件定义好这些文件、编译器指令等。在 `npm install` 执行的时候，`node-gyp` 就会被自动调用以编译你的 C++ 代码，并将动态链接库放入项目或者包根目录的 `build` 目录下。

GYP 的全称为 Generate Your Project，它是谷歌当年为 Chromium 写的构建工具。在当年被编写出来的时候，它旨在成为诸如 `autoconf`¹ 等工具的继承者（虽然现在 GN² 的势头貌似更猛）。

¹ <https://www.gnu.org/software/autoconf/>

² <https://chromium.googlesource.com/chromium/src/+/master/tools/gn/README.md>

GYP 通过读取特定的 JSON 配置来为项目产生相对应操作系统的构建文件（如 Makefile、Visual Studio 项目文件等）。

这个 JSON 可不是一个标准的 JSON 格式，它可以使用单引号表示字符串，甚至可以往里面加注释等。具体可以访问 GYP 官方网站¹进行了解。

而 node-gyp 自 Node.js 0.8 版本后，就成了其默认的构建工具，从此 Node.js 就与 GYP 结下了不解之缘。node-gyp 对 GYP 进行了封装，将你项目的 **binding.gyp** 与标准的 Node.js GYP 文件进行合并，从而得到相应的构建文件。

4.1.1 惊鸿一瞥

binding.gyp 本质上是一个 JSON 文件，其定义了各种编译相关的内容。

一个最简单的 binding.gyp 文件可以像下面这样：

```
{
  "targets": [
    {
      "target_name": "binding",
      "sources": [ "src/binding.cc" ]
    }
  ]
}
```

这个文件的含义如下：

- 目标动态链接库的源码是 src/binding.cc；
- 目标动态链接库编译的结果名是 binding.node。

4.1.2 binding.gyp 基础结构

最终的动态链接库通常会被放到 build/Release/ 目录下，所以在我们自己的 JavaScript 源码中如果需要引入这个 C++ 扩展，就会通过类似于 `require("../build/Release/binding")` 的形式进行载入。

每个 binding.gyp 文件都需要有一个 targets 数组，其中每个元素都定义了一个即将被 node-gyp 构建的 C++ 的模块。targets 数组中这些模块的对象必须包含一个 target_name 字段，这代表了模块名，其在编译后会被 <模块名>.node 命名。除此之外，每个对象中还需要

¹ <https://gyp.gsrc.io/>

有 `sources` 字段，用于告知 GYP 哪些文件将作为源码被进行编译，这些文件可以是 GYP 所能进行编译的 C 或者 C++ 的任意格式文件（如 `*.c`、`*.cc` 或者 `*.cpp` 等）。在日常开发中，读者还可能需要一个 `include_dirs` 字段的数组，用于告诉 GYP 这个数组内的目录将被作为编译时 `#include` 搜索用的目录。

本书推荐使用 NAN¹ 开发 C++ 扩展，用于面对各种不同版本的 Node.js 情况。用它进行开发的话就需要通过 `#include <nan.h>` 来将包的头文件引入。此时我们就用 `"<(node -e \"require('nan')\")\""` 这个字符串告知 GYP 启动一个 Node.js 进程来执行 `nan` 这个包，这个包将会返回它的头文件所在的目录。所以我们将这个字符串放入 `include_dirs` 数组中，就告知了 GYP 将 `NAN` 所在的目录作为项目构建时的一个头文件搜索目录，主要用于当 `#include <nan.h>` 时 `nan.h` 能被正常引入。

在知道了上面的一个基础结构之后，读者就能完成绝大多数 C++ 扩展构建文件的配置了。有心的读者可以打开随书代码各个目录下的 `binding.gyp` 文件来查看，看看是不是都有这些字段存在。

其实除了最简单的结构之外，读者还应该知道其他一些对开发有帮助的配置项。别忘了 `node-gyp` 的配置文件本质上还是一个 GYP 文件，所以读者还应该知道一些 GYP 相关的内容。

4.1.3 GYP 文件

一个 GYP 配置文件是一个可配置、可编程的类 JSON 文件，里面有一些理念需要读者厘清。

1. 元类型

在一个 GYP 配置文件中，K-V 对中的键值可以是以下一些类型。

- **字符串**：通过单引号或者双引号包含。
- **整数**：比较罕见，通常在布尔类型指定的时候，约定 1 为真，0 为假。
- **数组**：以中括号包含，与 JSON 一致。
- **对象**：以大括号包含，与 JSON 一致。

2. 注释

与 JSON 文件不同，GYP 文件可以引入注释。注释的形式是以 `#` 开头的，与 Bash 等语言的注释差不多。

如：

¹ Node.js 原生抽象帮助包，可参考 <https://github.com/nodejs/nan>。

```
{
  'egg': [
    'flower',
    'soup', # 这是注释
  ],
}
```

3. 路径相关

在 GYP 文件中，相当多的字符串键值被用作文件路径。其中这些键名的值会被用作文件路径：

- destination
- files
- include_dirs
- inputs
- libraries
- outputs
- sources
- mac_bundle_resources
- mac_framework_dirs
- msvs_cygwin_dirs
- msvs_props

除此之外，包含或者以下列字符串作为后缀的一些字段也将被视为文件路径：

- _dir
- _dirs
- _file
- _files
- _path
- _paths

上述字段的键值，除了常规性地解析其路径之外，还有几种特殊情况来供编程用：

- /：表示一个绝对路径，如 "include_dirs": ["/"]。
- \$：表示一个构建系统（如 Visual Studio）自身的变量值，如 "libraries": ["\$(SDKROOT)/some/dir"]。

- -：用于表示类似于 `-llib` 之类的情况，这个样例表示“`lib` 作为链接库搜索路径”。
- <、> 和 !：用于 GYP 的变量或者命令。

4. 变量与指令

(1) 展开时机

对于 GYP 中的变量或者命令来说，展开时机有两种阶段。

- **先阶段**：`conditions` 字段处理或者 < 开头的变量展开处于先阶段。
- **后阶段**：`target_conditions` 字段的处理或者 > 开头的变量展开及命令展开都处于后阶段。

下面举个简单的例子，如果读者有这样一个 GYP 配置：

```
{
  'target_defaults': {
    'target_conditions': [
      ['_type=="shared_library"', {'cflags': ['-fPIC']}],
    ],
  },
  'targets': [
    {
      'target_name': 'sharing_is_caring',
      'type': 'shared_library',
    },
    {
      'target_name': 'static_in_the_attic',
      'type': 'static_library',
    },
  ],
}
```

在样例的配置条件中，并没有预先定义好 `_type` 变量，所以 `_type` 无法在先阶段被展开。正如本节中描述的那样，`target_conditions` 字段中的变量将在后阶段被展开，所以在解析 `targets` 时有了各自的 `_type` 上下文之后，`_type` 才在后阶段被展开。

所以，这个配置在展开后的结果就会是这样的：

```
{
  'targets': [
    {
      'target_name': 'sharing_is_caring',
      'type': 'shared_library',
    },
  ],
}
```



```

    'cflags': ['-fPIC'],
  },
  {
    'target_name': 'static_in_the_attic',
    'type': 'static_library',
  },
]
}

```

(2) 变量

在 GYP 中主要有 3 种类型的变量。

- **预定义变量**：以全大写、下画线的形式命名。预定义变量是被 GYP 自动定义的，虽然这些变量可被覆盖，但并不推荐这么做。常见的预定义变量如下（更多预定义变量请参考 GYP 官方文档）。
 - OS：当前操作系统，值为 'linux'、'mac' 和 'win' 的其中一个。
 - INTERMEDIATE_DIR：编译的中间目录，仅保证对单个 target 有效。
 - SHARED_INTERMEDIATE_DIR：与 INTERMEDIATE_DIR 相对应，该目录对所有 target 有效。
- **用户变量**：用户变量可以在 variables 字段中被一一声明，且按照规范来说，通常使用全小写、下画线的形式命名。
- **自动变量**：所有的字符串型键名都会被当成自动变量处理，其变量名是键值加上 _ 前缀。例如如果有这样一个键值对 type: 'static_library'，那么一个叫 _type 的变量就会被自动生成，其值就是 'static_library'。

在变量的声明段 variables 中，一个变量的键名如果以 % 结尾，则表示这个变量是一个防覆盖变量，它不会覆盖那些已被声明的变量。

以上这些变量在使用过程中，根据展开时机的不同，可以使用两种不同的前缀来表示：

- 以 < 开头的展开表达式属于先展开式，将于先阶段被展开。
- 以 > 开头的展开表达式属于后展开式，将于后阶段被展开。

除了 < 和 > 两种前缀表示不同的展开时期之外，GYP 还使用 @ 来代表不同类型的变量。

- 不带 @ 的 <(VAR) 或者 >(VAR)
 - 如果 VAR 是一个字符串，那么这个表达式代表的就是一个字符串。
 - 如果 VAR 是一个数组，那么这个表达式展开就是以生成器指定的分隔符拼接的字符串，通常这个分隔符是空格。

- 带 @ 的 <@ (VAR) 或者 >@ (VAR)
- 该表达式必须在数组上下文中出现。
- 数组项必须是 <@ (VAR) 或者 >@ (VAR)。
- 如果 VAR 是一个数组，数组中的内容会被一一插入当前所在的数组中。
- 如果 VAR 是一个字符串，那么表达式展开就是以生成器指定的分隔符进行分割的数组，然后被一一插入当前所在的数组中。

(3) 指令

指令与变量类似，不过指令比变量更高级的是，这里不仅把指令中的变量展开，还让 GYP 去启动进程执行这条展开的指令，然后以指令的输出作为结果替换掉当前的键值。**指令是以 <! 或者 <!@ 开头的。**

与变量类似，<!@ 的行为也用于数组中。

指令还有一个神奇的特性，就是可以嵌套。举个例子来说，如果有一个配置如下：

```
'variables' : [
  'foo': '<!(echo Build Date <!(date))',
],
```

那么展开后的结果就会像这样：

```
'variables' : [
  'foo': 'Build Date 02:10:38 PM Fri Jul 24, 2017 -0700 PDT',
],
```

除此之外，指令还支持把一个个参数分割到数组中，用于在执行的时候将参数用引号括起来：

```
'variables' : [
  'files': '<!(["ls", "-l", "Filename With Spaces"])',
],
```

再回到 NAN 的问题中来，node -e 是执行参数后面的代码并直接输出，那么 node -e require('nan') 就相当于执行这个 require 函数。通过查看 nan 这个包的结果可知，上面的指令将会输出类似于 node_modules/nan/ 的结果。那么在 include_dirs 里面加上一条 <!(node -e "require('nan')") 的话，就相当于把 node_modules/nan 这个目录加到了 include_dirs 数组中。

5. 条件分支

条件分支在 GYP 中也分为先阶段分支和后阶段分支：

- 先阶段分支以 `conditions` 字段出现。
- 后阶段分支以 `target_conditions` 字段出现。

条件分支字段可以存在于配置文件的任意部分，它内部所代表的内容将被合并到最近的一个上下文中。它的本质还是一个数组，数组内的每个元素表示一个分支。

然而，它的每个元素其实还是一个只包含两个元素的数组。

① 元素 1 是一个字符串，表示条件表达式。它可以是这样的一些形式：

- 对于字符串来说，可以使用 `var=="value"` 或者 `var!="value"`。如 `'OS=="linux"'`，就代表这个分支在当前操作系统是 Linux 的时候会被触发。
- 对于整数来说，使用 `var==value`、`var!=value`、`var<value` 等整数比较的操作符来判断。如 `'chromium_code==0'`，就代表当 `chromium_code` 这个变量为 0 的时候才会触发这个分支。

② 元素 2 是一个对象，代表要被合并到最近的一个上下文中的内容。

下面举个简单的例子：

```
{
  'sources': [
    'common.cc',
  ],
  'conditions': [
    ['OS=="mac"', {'sources': ['mac_util.mm']}],
    ['OS=="win"', {'sources': ['win_main.cc']}], {'sources': ['posix_main.
cc']}]],
    ['OS=="mac"', {'sources': ['mac_impl.mm']}],
    ['OS=="win"', {'sources': ['win_impl.cc']}],
    {'sources': ['default_impl.cc']}
  ],
],
}
```

这段配置的意思如下：

- 如果操作系统是 macOS，sources 就变成了 `['common.cc', 'mac_util.mm', 'posix_main.cc', 'mac_impl.mm']`；

- 如果操作系统是 Windows, sources 就变成了 ['common.cc', 'win_main.cc', 'win_impl.cc'];
- 否则, 该字段就会是 ['common.cc', 'posix_main.cc', 'default_impl.cc']。

6. 数组过滤器

数组过滤器指的是那些键名以 ! 或者 / 结尾的字段:

- 键名以 ! 结尾是一个排除过滤器, 表示这里的键值将被从无后缀的同名字段中排除。
- 键名以 / 结尾是一个匹配过滤器, 表示通过正则表达式匹配出相应的结果, 然后以指定方式 (包含或者排除) 进行处理。

(1) 排除过滤器 (!)

排除过滤器主要用于删除原数组中的一些元素。下面举个例子, 如果读者需要根据操作系统的不同将 sources 字段中的一些源文件去掉, 就可以使用这样的配置:

```
{
  'sources': [
    'mac_util.mm',
    'win_util.cc',
  ],
  'conditions': [
    ['OS=="mac"', {'sources!': ['win_util.cc']}],
    ['OS=="win"', {'sources!': ['mac_util.cc']}],
  ],
}
```

(2) 匹配过滤器 (/)

匹配过滤器与排除过滤器类似, 其配置数组中的每一个元素都是一个有两个子元素的数组。在该元素的两个子元素中, 第一个元素是一个字符串, 只能是 "include" 或者 "exclude", 表示匹配到的内容将被包含进无后缀同名数组的内容或者从中排除; 第二个元素就是正则表达式字符串了。

匹配过滤器会在同级的排除过滤器之后进行处理, 这样就为将某些已排除的文件重新加进来提供了可能。

下面再来提供一个样例的配置供大家解读:

```
{
  'sources': [
    'io_posix.cc',
    'io_win.cc',
  ],
}
```

```

    'launcher_mac.cc',
    'main.cc',
    'platform_util_linux.cc',
    'platform_util_mac.mm',
  ],
  'sources/': [
    ['exclude', '_win\\.cc$'],
  ],
  'conditions': [
    ['OS!="linux"', {'sources/': [['exclude', '_linux\\.cc$']]},
    ['OS!="mac"', {'sources/': [['exclude', '_mac\\.cc|mm?$']]},
    ['OS=="win"', {'sources/': [
      ['include', '_win\\.cc$'],
      ['exclude', '_posix\\.cc$'],
    ]}],
  ],
],
}

```

当这个配置中的条件分支执行完成之后，根据操作系统的不同，会有不同的结果。

- 若操作系统是 Linux，则包含 ['io_posix.cc', 'main.cc', 'platform_util_linux.cc']。
- 若操作系统是 macOS，则包含 ['io_posix.cc', 'launcher_mac.cc', 'main.cc', 'platform_util_mac.mm']。
- 若是 Windows，则包含 ['io_win.cc', 'main.cc', 'platform_util_win.cc']。

4.1.4 常用字段

对于每个 target 来说，有一些常用字段供它们来定义该目标的各种编译行为。这些字段如果是数组的话，都能使用排除过滤器、匹配过滤器以及条件分支等对它们进行操作。

1. 预编译（-D 或者 /D）

使用 "defines" 字段来为目标添加预编译的宏，例如：

```

{
  'targets': [
    {
      'target_name': 'existing_target',
      'defines': [
        'FOO',
        'BAR=some_value',
      ],
    },
  ],
}

```

```
    },
  ],
},
```

这相当于给编译时添加了 `-DFOO -DBAR=some_value` 参数。

2. 头文件搜索路径（`-I` 或者 `/I`）

使用 `"include_dirs"` 字段来为目标添加头文件搜索路径，例如：

```
{
  'targets': [
    {
      'target_name': 'existing_target',
      'include_dirs': [
        '..',
        'include',
      ],
    },
  ],
},
```

这相当于在编译时添加了 `-I.. -Iinclude` 参数。

3. 库

使用 `"libraries"` 字段来为编译添加链接库，相当于 `-L` 等编译标识。例如：

```
"conditions": [
  ["OS==" "mac\"", {}],
  ["OS==" "linux\"", {
    "libraries": [
      "../libonsclient4cpp.a"
    ]
  }],
  ["OS==" "win\"", {
    "libraries": [
      "../ONSCClient4CPP.lib"
    ]
  }],
],
```

以上配置修改自笔者自研的 Aliyun ONS SDK。由于目前阿里云官方 C++ SDK 并未提供源码，而是提供了 Windows 下和 Linux（GCC 4.x）下的两个链接库，因此这里通过 `libraries` 字段将它们引进来。

4. 编译标识

开发者还能添加自己所需要的各种编译标识，如 G++ 中的 `-Werror` 就代表把所有的编译警告都转为错误。

编译标识在 GYP 中通过 `cflags` 字段定义。下面举个例子说明：

```
{
  'targets': [{
    'target_name': 'existing_target',
    'conditions': [
      ['OS=="win"', {
        'cflags': [
          '/WX',
        ],
      }, { # OS != "win"
        'cflags': [
          '-Werror',
        ],
      }],
    ],
  }],
}
```

5. 链接标识

除了编译标识，开发者还可以定义链接标识：

- 在 Linux 以及大多数的非 macOS 的 POSIX 系统下，使用 `ldflags` 定义链接标识。
- 在 macOS 下，使用 `xcode_settings` 定义链接标识。
- 在 Windows 下，使用 `msvs_settings` 定义链接标识。

不过这些用法在大家真的需要用到时候才有意义，通常读者并不需要使用它们。例如：

```
{
  'targets': [
    {
      'target_name': 'existing_target',
      'conditions': [
        ['OS=="linux"', {
          'ldflags': [
            '-pthread',
          ],
        }],
      ],
    },
  ],
}
```

```

    ['OS="mac"', {
      'xcode_settings': {
        'GCC_ENABLE_CPP_EXCEPTIONS': 'YES',
      },
    }],
    ['OS="win"', {
      'VCLinkerTool': {
        'GenerateDebugInformation': 'true',
      },
    }],
  ],
},
],
},

```

6. 编译类型

在 Node.js 中，一个扩展会被编译成动态链接库。

如果读者想让其中一个 target 以静态链接库的形式进行编译，就需要修改 type 字段了。该字段在 Node.js C++ 扩展开发中，通常使用以下两种值。

- shared_library: 一般动态链接库。
- static_library: 静态链接库。
- loadable_module¹: Node.js 可直接载入的 C++ 扩展动态链接库，为 binding.gyp 的默认类型。

7. 依赖

如果读者的代码中用了—个第三方的 C / C++ 代码，就要在 binding.gyp 中将这个库编译成静态链接库，使其能在我们的主 target 中被依赖，这样就需要用 **dependencies** 字段了。

这种场景的常用做法就是，先写一个 target 用于编译第三方库，并设置为相应的编译类型（通常是 static_library），然后在自己的主 target 中以 dependencies 字段将第三方库的目标名引进来。

例如：

```

'targets': [{
  'target_name': 'xmempool',
  'type': 'static_library',

```

¹ 该类型是 node-gyp 特有的。


```

    'sources': [ './deps/xmempool/xmempool.c' ]
  }, {
    'target_name': 'byakuren',
    'type': 'static_library',
    'sources': [
      './deps/byakuren/byakuren.c',
      ...
    ],
    'dependencies': [ 'xmempool' ]
  }, {
    'target_name': 'thmclrx',
    'dependencies': [ 'byakuren' ],
    'sources': [
      './src/thmclrx.cc',
      './src/common.cc'
    ]
  }
}]

```

这一段配置修改自笔者自己开发的一个主题色提取包 Thmclrx¹ 的 binding.gyp 文件，它依赖于笔者实现的 C 版主题色提取库 Byakuren²，并且 Byakuren 依赖于笔者自己实现的 C 版内存池 Xmempool³。

所以就有了 3 个 target，然后依次依赖的场景。

不知道大家是否还记得 3.6.3 节中关于拦截器的介绍，两个拦截器在本书中分别有两套随书代码，即“6. mapped property interceptor”和“7. indexed property interceptor”。

在这两套代码中，对于 CNode 社区的请求是通过 HTTP 同步请求得到的。而 Node.js 底层并没有这样的 API。所以在其中对于 HTTP 请求用了 GitHub 上开源的小型 HTTP 请求库 minihttp，而这个 minihttp 又依赖了 mbed TLS 来支持 HTTPS 请求。

这两个库的代码实际上被放到了随书代码的 deps 目录下。然后在这两套代码各自的 binding.gyp 中就有了这两个依赖的编译选项。

```

{
  "targets": [{
    "target_name": "mbedtls",
    "type": "static_library",
    "sources": [
      # mbedtls 待编译源文件
    ]
  }]
}

```

1 笔者自研的 Node.js 图片主题色提取包，可访问 <https://github.com/XadillaX/thmclrx> 获取更多内容。

2 笔者自研的 C 版图片主题色提取库，可访问 <https://github.com/XadillaX/byakuren> 获取更多内容。

3 笔者自研的 C 版内存池库，可访问 <https://github.com/XadillaX/xmempool> 获取更多内容。

```

],
"include_dirs": [
    "../deps/mbedtls/include",
    "../deps"
],
"defines": [
    # 预定义宏 MBEDTLS_CONFIG_FILE
    "MBEDTLS_CONFIG_FILE=\"mbedtls/configs/config-mini-tls1_1.h\""
]
}, {
    "target_name": "mbedx509",
    "type": "static_library",
    "sources": [
        # mbedx509 待编译源文件
    ],
    "include_dirs": [
        "../deps/mbedtls/include",
        "../deps"
    ],
    "defines": [
        # 预定义宏 MBEDTLS_CONFIG_FILE
        "MBEDTLS_CONFIG_FILE=\"mbedtls/configs/config-mini-tls1_1.h\""
    ]
}, {
    "target_name": "mbedcrypto",
    "type": "static_library",
    "sources": [
        # mbedcrypto 待编译源文件
    ],
    "include_dirs": [
        "../deps/mbedtls/include",
        "../deps"
    ],
    "defines": [
        # 预定义宏 MBEDTLS_CONFIG_FILE
        "MBEDTLS_CONFIG_FILE=\"mbedtls/configs/config-mini-tls1_1.h\""
    ]
}, {
    "target_name": "mapped_property_interceptor",
    "sources": [
        "../deps/minihttp/minihttp.cpp",
        "interceptor.cpp"
    ],
    "include_dirs": [
        "../deps/mbedtls/include"
    ],
    ],

```

```

    "defines": [ "MINIHTTP_USE_MBEDTLS" ],
    "dependencies": [
        "mbbedtls",
        "mbedx509",
        "mbedcrypto"
    ]
  }
}
}

```

现在再回过头来看这个 `binding.gyp` 是不是就豁然开朗了呢？

思考一下：其实这个 `binding.gyp` 可以被优化，其中 `MBEDTLS_CONFIG_FILE` 这个预定义宏出现了多次，而且有几个 `include_dirs` 也出现了多次。这些内容可以用变量的形式被声明和引用，那么动手试试看吧。

8. 复制

通过 `"copies"` 字段定义一个数组，数组内的内容将会被复制到指定位置。

还是 Aliyun ONS SDK 的例子，官方提供的 Windows 链接库是动态的。众所周知，Windows 使用动态链接库的话，需要该 `.dll` 文件处于一个合适的位置。

于是笔者的这个包就将用到的动态链接库 `ONSCClient4CPP.dll` 复制到 Node.js 扩展编译的输出目录下。配置如下：

```

"conditions": [
  ["OS==" + "win", {
    "copies": [{
      "destination": "<(module_root_dir)/build/Release/",
      "files": [ "<(module_root_dir)/src/third_party/lib/windows/
ONSCClient4CPP.dll" ]
    }]
  }]
]

```

9. 常用变量

在 `node-gyp` 中，除了 GYP 自带的一些预定义变量，它还定义了一些专用于 `binding.gyp` 使用的变量。

其一是来自 Node.js 的 `common.gypi`。关于 Node.js v6.9.4 的变量可以参考 <https://github.com/nodejs/node/blob/v6.9.4/common.gypi>。

其二是来自 `node-gyp` 的 `addon.gypi`。大家可以访问 <https://github.com/nodejs/node-gyp/blob/master/addon.gypi> 进行参考。

其三是 `node-gyp` 在运行过程中通过命令行参数添加的一些变量。比如在前面提到的 `<(module_root_dir)` 这个变量就是在 `node-gyp` 运行时通过命令行参数添加进来的，指的是当前模块的根目录。

在此列出一些常用的变量。

- `module_root_dir`: 模块根目录。
- `node_root_dir`: Node.js 一些文件的根目录（如头文件等）。
- `node_gyp_dir`: `node-gyp` 这个包的根目录。
- `node_lib_file`: 用于编译时的 Node.js 库文件。

以上这些常用变量，都在笔者对于 `node-gyp` 中的一个 Issue¹ 里有所体现。大家可以参考 Issue 来获取一些相关信息，或者也许某一天 `node-gyp` 会完善关于这些变量的文档，到时候我们就可以直接参考文档了。

4.1.5 小结

本节主要介绍了 `binding.gyp` 文件的定义，告诉读者有哪些黑科技可以让你的 `binding.gyp` 变得更强大、更灵活。

它主要还是继承了 GYP 的功能，包括：

- 数据结构；
- 字段名；
- 变量；
- 指令；
- 条件。

有了这样一个“磨刀工”，后面做 C++ 扩展开发的“砍柴工”的效率就会大幅度提高。

4.1.6 参考资料

[1] nodejs/node-gyp: Node.js native addon build tool: <https://github.com/nodejs/node-gyp>.

[2] NodeSchool#Going Native: <https://nodeschool.io/#goingnative>.

¹ Issue 地址: <https://github.com/nodejs/node-gyp/issues/1223>。

[3] GYP - User Documentation: <https://gyp.gsrc.io/docs/UserDocumentation.md>.

[4] GYP - Input Format Reference: <https://gyp.gsrc.io/docs/InputFormatReference.md>.

[5] “binding.gyp” files out in the wild: <https://github.com/nodejs/node-gyp/wiki/%22binding.gyp%22-files-out-in-the-wild>.

4.2 牛刀小试

本节将介绍 Node.js v6.9.4 文档中的入门样例。其难度应该会小于或等于第 3 章的 Chrome V8 样例。本章与第 3 章的侧重点不同：第 3 章的侧重点在于用样例来解释各种 Chrome V8 的内容，而本章的侧重点则在于 Node.js 方面。

所以大家在阅读本章的时候，看到在前面章节中讲到的一些概念也是很正常的。毕竟对于 Node.js 来说，Chrome V8 是核心，而往上就是胶水层，将 Chrome V8、libuv 等各种好东西黏起来。

4.2.1 又是 Hello World

让我们打开“13. hello world again”，看看里面的各个文件吧。

首先是构建配置文件 binding.gyp:

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "hello.cc" ]
    }
  ]
}
```

根据 4.1 节的姿势，我们马上可以知道这个扩展名为 addon，并且需要编译 hello.cc 得到。然后再将目光凝聚到 hello.cc:

```
#include <node.h>

namespace demo {

using v8::FunctionCallbackInfo;
```

```

using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void Method(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();
    args.GetReturnValue().Set(String::NewFromUtf8(isolate, "world"));
}

void init(Local<Object> exports) {
    NODE_SET_METHOD(exports, "hello", Method);
}

NODE_MODULE(addon, init)
}

```

这段代码是不是很熟悉？相信仔细阅读了第 3 章的你应该对这里的每个函数以及每句话的含义能够如数家珍。

首先是一个模块载入时执行的函数 `init` 通过 `NODE_MODULE(addon, init)` 被定义；然后在 `init` 函数中与之前的内容有一点不同，这里使用了一个宏 `NODE_SET_METHOD` 进行函数的设置；最后是 `Method` 函数中，在得到 `Isolate` 指针后，通过 `args.GetReturnValue().Set()` 使函数的返回值是一个 "world" 字符串。

正如前面所说，在 `init` 里面有一点不同，就是设置函数时使用了 `NODE_SET_METHOD` 这个宏。它的声明在 `src/node.h` 中：

```

// 由于曾经的版本中这是一个宏，因此函数名全大写
inline void NODE_SET_METHOD(v8::Local<v8::Template> recv,
                           const char* name,
                           v8::FunctionCallback callback) {
    v8::Isolate* isolate = v8::Isolate::GetCurrent();
    v8::HandleScope handle_scope(isolate);
    v8::Local<v8::FunctionTemplate> t = v8::FunctionTemplate::New(isolate,
                                                                callback);
    v8::Local<v8::String> fn_name = v8::String::NewFromUtf8(isolate, name);
    t->SetClassName(fn_name);
    recv->Set(fn_name, t);
}

// 由于曾经的版本中这是一个宏，因此函数名全大写

```

```

inline void NODE_SET_METHOD(v8::Local<v8::Object> recv,
                           const char* name,
                           v8::FunctionCallback callback) {
    v8::Isolate* isolate = v8::Isolate::GetCurrent();
    v8::HandleScope handle_scope(isolate);
    v8::Local<v8::FunctionTemplate> t = v8::FunctionTemplate::New(isolate,
                                                                    callback);

    v8::Local<v8::Function> fn = t->GetFunction();
    v8::Local<v8::String> fn_name = v8::String::NewFromUtf8(isolate, name);
    fn->SetName(fn_name);
    recv->Set(fn_name, fn);
}
#define NODE_SET_METHOD node::NODE_SET_METHOD

```

如你所见，这个宏展开后的调用函数等同于读者自己去设置一个函数到一个对象下面：

```

v8::Local<v8::FunctionTemplate> t = v8::FunctionTemplate::New(isolate,
Method);
v8::Local<v8::Function> fn = t->GetFunction();
v8::Local<v8::String> fn_name = v8::String::NewFromUtf8(isolate, "hello");
fn->SetName(fn_name);
exports->Set(fn_name, fn);

```

所以，这段 `hello.cc` 中的 C++ 代码等同于这样一段 JavaScript 代码：

```

function hello() {
    return "world";
}

exports.hello = hello;

```

依旧是老规矩，执行一遍代码试试看吧：

```

$ node-gyp rebuild
...
$ node
> const addn = require("./")
undefined
> addn.hello();
'world'

```

这里能通过 `./` 直接在 `require` 时得到 C++ 扩展的原因是 `package.json` 中定义了 C++ 扩展编译后的文件作为 `main` 字段。还记得 1.2.2 节中说的 `main` 字段吗？

感觉如何？阅读本章内容是不是觉得特别轻松？那是因为你花了“磨刀”的时间，“砍柴”自然就简单了。

4.2.2 函数参数

本节对应的是 3.7.9 节。函数的参数是通过这个类型的变量被传入的。打开“14. function parameters”，好好浏览一番吧。

这里的 binding.gyp 文件大同小异，主要看 addon.cc。

```
#include <node.h>

namespace demo {

using v8::Exception;
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::String;
using v8::Value;

// 实现 "add" 函数
void Add(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();

    // 判断参数的个数是否合法
    if (args.Length() < 2) {
        // 不合法则抛错
        isolate->ThrowException(Exception::TypeError(
            String::NewFromUtf8(isolate, "Wrong number of arguments")));
        return;
    }

    // 判断参数的类型是否合法
    if (!args[0]->IsNumber() || !args[1]->IsNumber()) {
        isolate->ThrowException(Exception::TypeError(
            String::NewFromUtf8(isolate, "Wrong arguments")));
        return;
    }

    // 计算第一个参数加第二个参数的 `double` 值
    // 并新生成一个 Local<Number> 句柄，将计算出来的 `double` 传入
```



```

double value = args[0]->NumberValue() + args[1]->NumberValue();
Local<Number> num = Number::New(isolate, value);

// 设置返回值为新生成的 `num`
args.GetReturnValue().Set(num);
}

void Init(Local<Object> exports) {
    NODE_SET_METHOD(exports, "add", Add);
}

NODE_MODULE(addon, Init)
}

```

该有的注释都已经写在代码中了，这里就不多加解释了。相信大家看到这里已经能把第 3 章中所学的零碎 API（包括容错的一些处理）以生产可用的方式组织起来了。

这段代码中比较重要的思想就是在运算的时候事先判断一下参数的合法性，不合法就往上一层抛错供其捕获。这样就不至于在出现不合法内容的时候，直接在 Chrome V8 层面发生崩溃导致进程退出了。

继续测试一下这段代码吧。

```

$ node-gyp rebuild
...
$ node
> const addon = require("./");
undefined
> addon.add(1, 2);
3
> addon.add(NaN, 2);
NaN
> addon.add("", 2);
TypeError: Wrong arguments
    at TypeError (native)
    at repl:1:7
    at sigintHandlersWrap (vm.js:96:12)
    at REPLServer.defaultEval (repl.js:313:29)
    at bound (domain.js:280:14)
    at REPLServer.<anonymous> (repl.js:513:10)
    at emitOne (events.js:101:20)
    at REPLServer.emit (events.js:188:7)
    at REPLServer.Interface._onLine (readline.js:239:10)
    at REPLServer.Interface._line (readline.js:585:8)

```

其中最后一条测试是输入了非法内容，在 C++ 扩展的参数校验一步中探测到了不合法的数据类型，就抛出了一个错误。

4.2.3 回调函数

还记得 Chrome V8 函数对象吗？虽然它是一个对象，但却能被调用。本节侧重于介绍函数调用实战的姿势。打开“15.run callback”的 addon.cc 文件。

```
#include <node.h>

namespace demo {

using v8::Function;
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Null;
using v8::Object;
using v8::String;
using v8::Value;

void RunCallback(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();
    Local<Function> cb = Local<Function>::Cast(args[0]);
    const unsigned argc = 1;
    Local<Value> argv[argc] = { String::NewFromUtf8(isolate, "hello world") };
    cb->Call(Null(isolate), argc, argv);
}

void Init(Local<Object> exports, Local<Object> module) {
    NODE_SET_METHOD(module, "exports", RunCallback);
}

NODE_MODULE(addon, Init)
}
```

在这里的 RunCallback 函数中，获取了第一个参数并将其转为函数，然后将唯一的参数 "hello world" 传给这个函数执行。下面执行这段代码看看：

```
$ node-gyp rebuild
...
$ node
> const addon = require("./")
```

```

undefined
> addon(function(msg) { console.log(msg); });
hello world
undefined
> addon(123);
[1] 25885 segmentation fault node

```

当笔者传入 `function(msg) { console.log(msg); }` 这个函数时，由 C++ 扩展传进来的 "hello world" 就被终端输出了。而在第二条测试的时候，居然又从 V8 层面崩溃了！想来也是，当传入的参数不是函数的时候，我们进行数据类型强制转换，并且强行调用的确会导致它崩溃。

这个时候我们就应该在 C++ 代码上写上判断函数等一系列异常处理代码吗？别着急，笔者觉得这么做太麻烦了。

那应该怎么办呢？笔者个人认为，在 C++ 层面做这些错误处理的麻烦程度远大于在 JavaScript 层面做。所以，笔者觉得这些事情应该交给 JavaScript 来做。

怎么做呢？不妨再给你的包写一个 JavaScript 文件吧，然后把 `package.json` 中的 `main` 指向该 JavaScript 文件，并且在文档中也不给出关于 C++ 扩展的调用方式。

下面看看这个目录下的 `addon.js` 吧：

```

"use strict";

const addon = require("../build/Release/addon");

module.exports = function(callback) {
  if(!callback || typeof callback !== "function") {
    throw new Error("Wrong argument");
  }

  addon(callback);
};

```

如你所见，笔者对这个函数重新包装了一番，做了一些参数上的非法判断。当一切合法后，再调用真正的 C++ 扩展。

关于这些异常的处理情景，读者可以按照自己的喜好来写。读者如果觉得把所有参数判断写到 C++ 代码中更方便，也可以这么做。

最后再来运行 Node.js 的 REPL 吧：

```

$ node
> const addon = require("./addon")
undefined
> addon(function(msg) { console.log(msg); });
hello world
undefined
> addon(123);
Error: Wrong argument
    at module.exports (/Users/USER/15. run callback/addon.js:12:15)
    at repl:1:1
    at sigintHandlersWrap (vm.js:96:12)
    at REPLServer.defaultEval (repl.js:313:29)
    at bound (domain.js:280:14)
    at REPLServer.<anonymous> (repl.js:513:10)
    at emitOne (events.js:101:20)
    at REPLServer.emit (events.js:188:7)
    at REPLServer.Interface._onLine (readline.js:239:10)
    at REPLServer.Interface._line (readline.js:585:8)
> addon(null);
Error: Wrong argument
    at module.exports (/Users/USER/15. run callback/addon.js:12:15)
    at repl:1:1
    at sigintHandlersWrap (vm.js:96:12)
    at REPLServer.defaultEval (repl.js:313:29)
    at bound (domain.js:280:14)
    at REPLServer.<anonymous> (repl.js:513:10)
    at emitOne (events.js:101:20)
    at REPLServer.emit (events.js:188:7)
    at REPLServer.Interface._onLine (readline.js:239:10)
    at REPLServer.Interface._line (readline.js:585:8)

```

第一条测试正常执行；后面两条测试由于参数不合法，也能正常地抛出错误了，而不是 V8 直接崩溃。

对象返回

对象的使用方法无非是 Set、Get 等一系列函数，所以也没什么特别的。本节样例的函数会根据参数传进来的内容，将其挂载到被返回的对象的 msg 字段中。

大家打开“16. object factory”的 addon.cc 阅读一下吧。

```

#include <node.h>

namespace demo {

```

```

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();

    Local<Object> obj = Object::New(isolate);
    obj->Set(String::NewFromUtf8(isolate, "msg"), args[0]->ToString());

    args.GetReturnValue().Set(obj);
}

void Init(Local<Object> exports, Local<Object> module) {
    NODE_SET_METHOD(module, "exports", CreateObject);
}

NODE_MODULE(addon, Init)
}

```

想必大家也都看到了，在 `CreateObject` 函数中，一个新的空对象 `obj` 被创建，然后通过 `obj->Set` 将传入的第一个参数转换成字符串挂到它的 `msg` 字段下，最后返回这个对象。

接下来执行一下刚才完成的代码：

```

$ node-gyp rebuild
...
$ node
> const addon = require("./");
undefined
> addon("蛋花汤");
{ msg: '蛋花汤' }
> addon("南瓜饼");
{ msg: '南瓜饼' }
> addon(null);
{ msg: 'null' }
> addon(NaN);
{ msg: 'NaN' }
> addon(function() {});
{ msg: 'function () {}' }

```

看到运行结果大家就明白了，所有传入的参数都会被转成字符串，然后被挂载到一个新对象的 `msg` 下。

4.2.4 函数返回

这回变一个花样，在“17. function factory”中打开 `addon.js` 看看需要实现怎样的功能。

```
const addon = require("./");

const fn = addon();
console.log(fn());
```

从 C++ 扩展中拿到的内容是一个函数，而通过调用这个函数 `addon()` 又返回了一个函数 `fn`。最后输出调用 `fn()` 的结果。

其实这种代码也很简单，实例化一个函数的事情我们在很早之前就已经知道了。各种 `Init` 里面不都是这么做的吗？这里无非是从把 `Init` 里面实例化的函数挂在到 `exports` 下，改成将返回值设置成实例化的函数。

下面来看看 `addon.cc`。

```
#include <node.h>

namespace demo {

using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void MyFunction(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();
    args.GetReturnValue().Set(String::NewFromUtf8(isolate, "hello world"));
}

void CreateFunction(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();

    Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, MyFunction);
    Local<Function> fn = tpl->GetFunction();
}
```

```

// 设置函数名为 `theFunction`
fn->SetName(String::NewFromUtf8(isolate, "theFunction"));

args.GetReturnValue().Set(fn);
}

void Init(Local<Object> exports, Local<Object> module) {
    NODE_SET_METHOD(module, "exports", CreateFunction);
}

NODE_MODULE(addon, Init)
}

```

`module.exports` 导出了 `CreateFunction` 这个函数。在这个函数中通过一个函数模板去封装 `MyFunction` 并实例化，在改了函数名之后将其返回。而 `MyFunction` 中就是简简单单地返回一个 "hello world" 字符串。

下面运行一下这个 `addon.js` 吧：

```

$ node-gyp rebuild
...
$ node addon.js
hello world
$ node
> const addon = require("./")
undefined
> const fn = addon();
undefined
> fn.name
'theFunction'
> fn.toString();
'function theFunction() { [native code] }'

```

4.2.5 小结

本节基于前面介绍的 Chrome V8 知识，讲解了最基础的几个 Node.js C++ 扩展样例：

- 函数参数；
- 回调函数；
- 对象返回；
- 函数返回。

这几个样例实际上都是 Node.js 官方文档中的实例。读者可能以前在看文档的时候还觉得不太明白，不过笔者相信大家阅读了本书前面的一些章节之后，就能很轻松地理解这些代码在各个方面的用途了。

4.2.6 参考资料

[1] Addons | Node.js v6.9.4 Documentation: <https://nodejs.org/docs/v6.9.4/api/addons.html>.

4.3 循序渐进

4.2 节基本上帮助读者把 V8 的知识简单串联了一遍。本节将会深入一层，讲解如何用 C++ 封装出一个 JavaScript 类，并且其中包含了 C++ 自有的一些底层数据结构——这些数据结构在 JavaScript 层面是无感知的。

聪明的读者可能发现了，这不就是对象模板的内置字段吗？对了一半，Node.js 为这种对象封装专门搞出了一个新的姿势——ObjectWrap 类。我们就来看看怎么使用它吧。

4.3.1 C++ 与 JavaScript 类封装

如果大家是“空降”¹到本节的话，推荐先阅读 3.7.5 节关于 C++ 内置字段的相关内容。因为 C++ 与 JavaScript 类的封装涉及内置字段。

首先在这里介绍一下 Node.js 为 JavaScript 类封装而提供的一个帮助类——ObjectWrap，它在 Node.js 的源码头文件 `src/node_object_wrap.h` 中。当然它也存在通过 `node-gyp` 安装后的本地 Node.js 头文件目录中。

ObjectWrap 的成员函数如下：

```
class ObjectWrap {
public:
    ObjectWrap();
    virtual ~ObjectWrap();

    template<class T>
    static inline T* Unwrap(Local<Object> handle);
```

¹ 二次元术语，指直接将动漫视频的进度条拖放到刚好跳过片头的位置，直接进入正片。在这里表示“跳过先前章节，直接翻到这一页”。


```

inline Local<Object> handle();
inline Local<Object> handle(Isolate* isolate);
inline Persistent<Object>& persistent();

protected:
    inline void Wrap(Local<Object> handle);
    inline void MakeWeak();
    virtual void Ref();
    virtual void Unref();

private:
    static void WeakCallback(const WeakCallbackInfo<ObjectWrap>& data);
};

```

若读者自己进行开发，通常是先写一个 C++ 类，继承自 `ObjectWrap`，然后做一些自己爱做的事情。下面先来讲一下几个比较常用的 API。

- `void Wrap(Local<Object> handle)`：将传入的 `Object` 本地句柄弄成一个与当前 `ObjectWrap` 对象关联的对象，即设置内置字段。
- `T* Unwrap(Local<Object> handle)`：静态函数，从 `Object` 本地句柄中获取与之关联的 `ObjectWrap` 对象。

两个函数一设一取，就形成了一个非常好的生态。

1. `MyObject` 的构造函数与原型链

笔者先设计一个 `JavaScript` 类，它有一个 `plusOne` 的成员函数，是把内部的值加一并返回。如果是用 `JavaScript` 写的，可能是下面这样的：

```

function MyObject(value) {
    this.value_ = value || 0;
}

MyObject.prototype.plusOne = function() {
    return ++this.value_;
};

```

代码的意思显而易见，与笔者之前描述的一致。那么如果想把这个 `MyObject` 交予 C++ 实现，就应该先声明一个 `MyObject` 的类，然后继承自 `ObjectWrap`。

```

class MyObject : public ObjectWrap {
};

```

构造函数中有一个参数，是初始的 `value_` 值。然后写一个静态函数，为 `exports` 挂上自己这个类的构造函数。

```
class MyObject : public ObjectWrap {
public:
    static void Init(Local<Object> exports);

protected:
    explicit MyObject(double value = 0);
    ~MyObject();

    double value_;
};

MyObject::MyObject(double value) : value_(value)
{
}

MyObject::~~MyObject()
{
}

void MyObject::Init(Local<Object> exports)
{
    Isolate* isolate = exports->GetIsolate();

    // 准备构造函数的函数模板
    Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, 构造函数);
    tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));

    // 因为通过函数模板实例化出来的类生成的对象要与 `MyObject` 绑定，
    // 所以要为其预留一个内置字段的位置。
    tpl->InstanceTemplate()->SetInternalFieldCount(1);

    // 设置原型链
    ...

    // 得到实例化函数并返回
    exports->Set(String::NewFromUtf8(isolate, "MyObject"), tpl-
>GetFunction());
}
```

这样一来，笔者在最外面一层的 `Init` 函数中只要再调用一下这个对象的静态函数，就能把对象的构造函数挂载到 `exports` 下了：

```
void InitAll(Local<Object> exports) {
    MyObject::Init(exports);
}

NODE_MODULE(addon, InitAll)
```

有了 MyObject 这个 C++ 类的构造函数之后，我们还要为导出的 JavaScript 类写一个构造函数实体，供 FunctionTemplate 去封装。同样地，我们将这个 JavaScript 类用的构造函数实体以 MyObject 静态函数的形式书写。

```
class MyObject : public ObjectWrap {
public:
    static void Init(Local<Object> exports);

protected:
    explicit MyObject(double value = 0);
    ~MyObject();

    // 与其他的 JavaScript 函数一样，都有 `FunctionCallbackInfo` 参数
    static void New(const FunctionCallbackInfo<Value>& args);

    double value_;
};

void MyObject::Init(Local<Object> exports)
{
    ...

    Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate,
MyObject::New);
    tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));

    ...
}

void MyObject::New(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();

    // 从构造函数中取出第一个参数，作为初始的 `value`
    double value = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();

    // 创建一个 `MyObject` 对象，并通过 `Wrap()` 函数将其与 `args.This()` 对象联立
    // 起来
    MyObject* obj = new MyObject(value);
```

```
obj->Wrap(args.This());

// 最后返回 this
args.GetReturnValue().Set(args.This());
}
```

这段代码理解起来就是这样的：

- ① 进入 Init 函数做初始化工作。
- ② 在 Init 中用一个函数模板把 `MyObject::New` 这个构造函数实例化出来。
- ③ 通过 `exports->Set` 导出构造函数。

而 `New` 构造函数所做的事如下：

- ① 获取参数传入的初始 `value` 值，若不传入则默认为 0。
- ② 将该 `value` 当作参数去新建一个 `MyObject` 对象。
- ③ 将新建出来的 `MyObject` 对象与 `this` 通过 `Wrap()` 关联起来。

这段代码整体来说不难理解。不过有几点要注意一下：首先，由于 `ObjectWrap::Wrap` 函数是 `protected` 的，因此调用它的时候必须是在它的对象或者子对象中，这就是笔者把 `New` 作为 `MyObject` 的静态函数的原因了。其次，由于我们只需要通过 `MyObject::Init` 来初始化整个过程，其他所有函数对外部都是不暴露的，因此这个类中也仅有 `Init` 函数是 `public` 的。

在 JavaScript 的构造函数完工之后，我们还需要在其原型链上挂载一个 `plusOne` 函数。这相对来说就简单多了。不过为了能调用 `Unwrap` 函数，这个 `plusOne` 仍需要放到 `MyObject` 的静态函数中。

```
class MyObject : public ObjectWrap {
public:
    static void Init(Local<Object> exports);

private:
    explicit MyObject(double value = 0);
    ~MyObject();

    static void New(const FunctionCallbackInfo<Value>& args);
    static void PlusOne(const FunctionCallbackInfo<Value>& args);
    double value_;
};

void MyObject::Init(Local<Object> exports)
```

```

{
    ...

    Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate,
MyObject::New);
    tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));

    ...

    // 设置原型链
    NODE_SET_PROTOTYPE_METHOD(tpl, "plusOne", PlusOne);

    ...
}

void MyObject::PlusOne(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();

    MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.Holder());
    obj->value_ += 1;

    args.GetReturnValue().Set(Number::New(isolate, obj->value_));
}

```

通过代码我们了解到，在 `Init` 函数中，通过 `NODE_SET_PROTOTYPE_METHOD` 宏给 `tpl` 这个函数模板的原型链加上了 `PlusOne` 函数。

`NODE_SET_PROTOTYPE_METHOD` 宏在这里的作用类似于下面这样：

```

Local<FunctionTemplate> t = FunctionTemplateNew(isolate, PlusOne);
Local<String> name = String::NewFromUtf8(isolate, "plusOne");
t->SetClassName(name);
tpl->PrototypeTemplate()->Set(name, t);

```

上面是一个不严谨的展开形态，有兴趣的读者可以自行去看该宏的一个完全展开形态。

至此为止，`Init` 函数的内容更加明晰了：新建一个构造函数 `New`，然后为其加上原型链上的函数 `PlusOne`，最后将这个构造函数返回。

不过读者可能要发问了：这里怎么没有看到 `MyObject` 析构函数呢？就算先前章节把内置字段与对象绑定的时候，也要通过一个弱持久句柄来维护它们的生命周期，这里怎么没有呢？

其实这些都是有的，早就被 `ObjectWrap` 封装好了。所以开发者在这个层面根本不需要关心它们的生命周期，一旦它们通过 `Wrap` 函数绑定在一起，生命周期也就绑定在一起了。

好了，本节所讲代码的最终形态就在随书代码的“18. myobject/1”下面，大家可以再串联起来阅读。之后就可以运行一下了。

```
$ node-gyp rebuild
...
$ node
> const addon = require("./build/Release/addon1");
undefined
> const a = new addon.MyObject(232);
undefined
> a
MyObject {}
> a.plusOne();
233
> a.plusOne();
234
```

如终端输出所示，通过 `new` 执行构造函数之后，我们就得到了它的一个实例 `a`。输出 `a` 看结果，我们发现没有任何值，因为它已经被隐藏在内置对象中了。然后执行 `a.plusOne()`，就会得到它内部 `value_` 自增的一个结果，依次会变成 233、234……

但是大家再尝试一下这样的运行代码吧：

```
const addon = require("./build/Release/addon1");
undefined
> const a = addon.MyObject(232);
Assertion failed: (handle->InternalFieldCount() > 0), function Wrap, file /
Users/USER/.node-gyp/6.9.1/include/node/node_object_wrap.h, line 56.
[1] 57948 abort node --harmony
```

看！崩溃了！为什么呢？

其实也好理解，在执行 `new addon.MyObject(232)` 的时候，`this` 是这个构造函数本身，它既是函数也是对象（在 JavaScript 中可以将它们等同理解），而构造函数本身是通过一个被设置内置字段数的函数模板实例化而来的，它有一个内置字段槽，在执行到 `obj->Wrap` 的时候就能把自身绑定进去。而在不通过 `new` 执行的时候，就是一个普通的函数调用，这个时候的 `this` 就是当前上下文，它并没有经过特殊设置，也没有内置字段的位置，于是就崩溃在了 `obj->Wrap` 中了。

如果想让自己的模块更能容错，可以阅读接下来的内容，使实例化对象的时候也可以不通过 `new` 的形式而直接调用函数得到。

2. 不通过 new 的实例化方法

老一辈的 JavaScript 程序员经常会有不用 new 实例化某个类的对象的习惯。例如：

```
new String("123");
// vs
String("123");

new Date();
// vs
Date();
```

笔者刚写的 `MyObject`，如果用这种方式调用函数的话，进程直接就崩溃了。现在告诉大家如何在自己的代码中加一点魔法，使得类实例化的时候也可以通过没有 new 的形式生成。

首先，写一个构造函数的持久句柄，作为 `MyObject` 的静态成员存在。下面先来看一下 `myobject.h` 的修改：

```
class MyObject : public ObjectWrap {
public:
    static void Init(Local<Object> exports);

private:
    explicit MyObject(double value = 0);
    ~MyObject();

    static void New(const FunctionCallbackInfo<Value>& args);
    static void PlusOne(const FunctionCallbackInfo<Value>& args);
    static v8::Persistent<v8::Function> constructor;
    double value_;
};
```

然后在 `MyObject::Init` 初始化的时候，除了将本地函数句柄返回之外，另外再将构造函数赋予这个持久句柄。下面看一下 `myobject.cc` 的改动。

```
Persistent<Function> MyObject::constructor;

void MyObject::Init(Local<Object> exports)
{
    Isolate* isolate = exports->GetIsolate();

    Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
    tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));
    tpl->InstanceTemplate()->SetInternalFieldCount(1);
```

```

NODE_SET_PROTOTYPE_METHOD(tpl, "plusOne", PlusOne);

// 就在这里将 tpl->GetFunction() 赋予持久句柄, 可以在别的地方被使用
constructor.Reset(isolate, tpl->GetFunction());
exports->Set(String::NewFromUtf8(isolate, "MyObject"), tpl->GetFunction());
}

```

有了这个构造函数的持久句柄之后, 我们就能开始做事情了。

还记得笔者先前讲的 `FunctionCallbackInfo` 参数吗? 它有一个 `IsConstructorCall()` 的成员函数, 代表这次调用是否是 `new` 的构造函数调用。笔者只需在这里做一些修改, 判断一下该次调用是否是构造函数调用。若是构造函数调用, 则走之前写好的逻辑; 若不是, 则手动给它构造一个对象并返回。

下面继续看看 `myobject.cc` 的其他改动吧。

```

void MyObject::New(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();

    if(args.IsConstructCall()) {
        // 如果是 `new` 构造函数调用, 则逻辑不变
        double value = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
        MyObject* obj = new MyObject(value);
        obj->Wrap(args.This());
        args.GetReturnValue().Set(args.This());
    } else {
        // 如果不是 `new` 构造函数调用, 则通过持久化构造函数句柄生成
        const int argc = 1;
        Local<Value> argv[argc] = { args[0] };
        Local<Context> context = isolate->GetCurrentContext();
        Local<Function> cons = Local<Function>::New(isolate, constructor);
        Local<Object> result =
            cons->NewInstance(context, argc, argv).ToLocalChecked();
        args.GetReturnValue().Set(result);
    }
}

```

在非 `new` 构造函数调用的分支中, 先新建了一个参数数组, 里面是唯一的参数 `args[0]`。然后再从 `constructor` 这个持久化句柄中获取到本地构造函数句柄 `cons`, 我们在先前的 `Init` 函数中已经将这个持久化句柄与 `New` 函数绑定了。

下一步就直接通过 `cons->NewInstance()` 函数来调用这个构造函数。这个函数笔者也在 3.7.6 节中提到过。

在通过 `NewInstance()` 得到实例化对象后，再将该对象返回即可。

本节的样例代码在“18. myobject/2”中，大家去翻阅一下完整代码吧，然后再实际运行一下。

```
$ node-gyp rebuild
...
$ node
> const addon = require("../build/Release/addon2")
undefined
> const a = new addon.MyObject(232);
undefined
> a.plusOne();
233
> const b = addon.MyObject(232);
undefined
> b.plusOne();
233
```

怎么样，两种调用方式都会了吧？

3. 将 `PlusOne` 抽象出去

看到这里之后，也许读者又有问题了：明明是一个 `MyObject` 类，为什么就净是一些静态函数呢？用一个类来表示这些东西是不是都是摆设呢？

其实这个类中有很多已经帮你处理好的内容了，对读者来说这都是如黑盒一样存在的，所以它看起来并不像一个传统的类。事实上，在稍微复杂一点的项目中，成员函数还是应该被抽象出来的。

如 `PlusOne` 函数，理想状态是通过直接调用 `obj->PlusOne()` 来实现的，而不是直接执行 `obj->value_ += 1`。

其实这就是一个抽象与否的问题。动动手就能直接把 `PlusOne` 给抽象出去，不过这个静态函数 `PlusOne` 是不能被抹除的——它是实例化原型链上函数的必要填充物。

下面看一下“18. myobject/3”里面的代码，实际上就是把 `obj->value_ += 1` 一步抽象成了 `MyObject` 的一个成员函数而已。

```
class MyObject : public ObjectWrap {
public:
    ...
```

```

    double PlusOne();

protected:
    static void PlusOne(const v8::FunctionCallbackInfo<v8::Value>& args);
    ...
};

void MyObject::PlusOne(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();

    MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.Holder());
    args.GetReturnValue().Set(Number::New(isolate, obj->PlusOne()));
}

double MyObject::PlusOne()
{
    this->value_ += 1;
    return this->value_;
}

```

在阅读完目录里面的源码后，仍然回过头来执行一遍看看结果。

```

$ node-gyp rebuild
...
$ node
> const addon = require("./build/Release/addon2")
undefined
> const a = new addon.MyObject(232);
undefined
> a.plusOne();
233
> const b = addon.MyObject(232);
undefined
> b.plusOne();
233

```

其结果还是与先前的一模一样。

4.3.2 实例化 C++ 类封装对象的函数

其实本节的内容与 4.3.1 节中不通过 `new` 进行实例化方法的原理差不多。首先我们来看看与之类似的 JavaScript 代码：

```
function MyObject(value) {
    this.value_ = value || 0;
}

MyObject.prototype.plusOne = function() {
    return ++this.value_;
};

exports.createObject = function(initValue) {
    return new MyObject(initValue);
};
```

笔者在本节中讲的就是如何实现这个 `createObject` 函数。反应快的读者应该已经想到了，换汤不换药，还是使用 `NewInstance` 函数。

没错，笔者为 `MyObject` 补上一个静态函数，用于实例化一个对象。

```
class MyObject : public ObjectWrap {
public:
    ...
    static void NewInstance(const FunctionCallbackInfo<Value>& args);
};

void MyObject::NewInstance(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();

    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Context> context = isolate->GetCurrentContext();
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    Local<Object> result =
        cons->NewInstance(context, argc, argv).ToLocalChecked();
    args.GetReturnValue().Set(result);
}
```

这个函数看起来跟 `New` 函数的非构造函数调用分支一模一样——毕竟都是通过非 `new` 形式调用函数实例化一个对象。

接着把这个函数通过 `NODE_SET_METHOD` 宏在 `addon.cc` 文件中挂载上去：

```
void InitAll(Local<Object> exports, Local<Object> module)
{
    ...
```

```

    NODE_SET_METHOD(module, "exports", MyObject::NewInstance);
}

```

这样就大功告成了。是不是非常简单？好的，接下来让我们进入更深层次的思考。如果笔者想让这个类只通过这个函数进行实例化，而无法在外部 JavaScript 逻辑中获取构造函数，更不能让其在外部通过 `new` 来分配这个对象，该如何隐藏这个构造函数呢？

大家随便删减代码其实就能做到，不过这里介绍一个更优雅的方式。

首先使 `MyObject::Init` 不再对 `exports` 对象挂载构造函数，不过还是需要设置一个持久句柄供 `NewInstance` 使用。那么这时，传进 `MyObject::Init` 的参数就无须是该 `exports` 了，一个 `Isolate` 隔离实例就够了。

```

class MyObject : public ObjectWrap {
public:
    static void Init(Isolate* isolate);
    ...
};

void MyObject::Init(Isolate* isolate)
{
    Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
    tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));
    tpl->InstanceTemplate()->SetInternalFieldCount(1);

    NODE_SET_PROTOTYPE_METHOD(tpl, "plusOne", PlusOne);

    constructor.Reset(isolate, tpl->GetFunction());
}

```

相应地，我们需要在 `addon.cc` 这个源文件的 `InitAll` 函数中，改变 `MyObject::Init` 的调用方式：

```

void InitAll(Local<Object> exports, Local<Object> module)
{
    MyObject::Init(exports->GetIsolate());
    NODE_SET_METHOD(module, "exports", MyObject::NewInstance);
}

```

这么乍一看，代码确实干净了不少，但实际上还有一个地方可以精简。我们既然已经把类的构造函数隐藏在了 C++ 代码中不暴露给 JavaScript，那么也就是说，使用者在调用的时候不

可能出现以非 `new` 的形式对构造函数进行调用，这样一来在 `New` 函数中，以非 `new` 的形式调用条件分支已经没用了。让我们再精简一下代码吧。

```
void MyObject::New(const FunctionCallbackInfo<Value>& args)
{
    double value = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
}
```

于是这个函数又变回了它最初的样子，就跟“18. myobject/4”中如出一辙，大家快去看看吧。

改造完成后，读者就可以测试一下自己的代码了。

```
$ node-gyp rebuild
...
$ node
> const createObject = require("../build/Release/addon4")
undefined
> const a = createObject(232);
undefined
> a.plusOne();
233
> a.plusOne();
234
```

4.3.3 将 C++ 类封装对象传来传去

在 4.3.1 节和 4.3.2 节中，笔者讲述了 `ObjectWrap` 的各种玩法，现在我们继续来加一点料吧。想象一下这样的一份 JavaScript 代码改造：

```
function MyObject(value) {
    this.value_ = value || 0;
}

MyObject.prototype.plusOne = function() {
    return ++this.value_;
};

exports.createObject = function(initValue) {
    return new MyObject(initValue);
};
```

```
exports.add = function(obj1, obj2) {
  // 读者可自行考虑容错
  return obj1.value_ + obj2.value_;
}
```

大家应该都看见了，add 函数将两个 MyObject 对象的 value_ 相加并返回。那么这样的代码放在我们的 C++ 扩展中要怎么实现呢？让我们继续改造 MyObject 的代码吧。

就目前的 MyObject 代码来看，由于 value_ 成员变量是私有的，在外部无法获取它的值，因此首要的任务就是为这个变量提供一个对外的接口。

```
class MyObject : public ObjectWrap {
public:
  inline double value() const { return value_; }

  ...
};
```

有了这个后门之后，就可以开心地开始 Add 函数的编写了。其思想很简单，就跟图 4-1 一样。

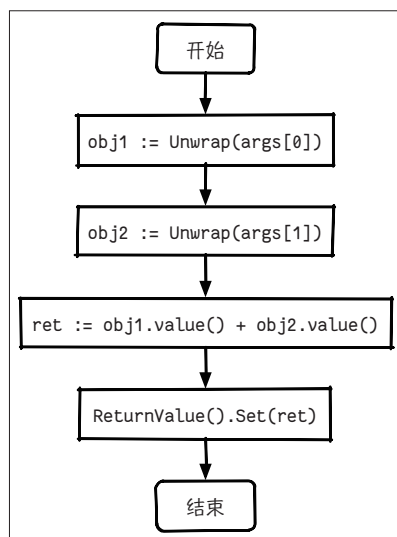


图 4-1 MyObject Add 流程图

跟着这个流程图，我们很容易就得出了这个“18. myobject/5/addon.cc”中 Add 函数的代码。

```
#include "myobject.h"

...
```

```

void Add(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();

    MyObject* obj1 = ObjectWrap::Unwrap<MyObject>(args[0]->ToObject());
    MyObject* obj2 = ObjectWrap::Unwrap<MyObject>(args[1]->ToObject());

    double sum = obj1->value() + obj2->value();
    args.GetReturnValue().Set(Number::New(isolate, sum));
}

void InitAll(Local<Object> exports, Local<Object> module)
{
    MyObject::Init(exports->GetIsolate());

    NODE_SET_METHOD(exports, "createObject", MyObject::NewInstance);
    NODE_SET_METHOD(exports, "add", Add);
}

```

跟流程图一模一样，先将两个 `MyObject` 通过 `ObjectWrap::Unwrap` 抽出来，然后再通过 `ObjectWrap::value()` 获取两个值并相加，最后返回这个相加的结果。

等着编译“18. myobject”并测试一遍吧。

```

$ node-gyp rebuild
...
$ node
> const addon = require("./build/Release/addon5");
undefined
> const a = addon.createObject(233);
undefined
> const b = addon.createObject(2100);
undefined
> addon.add(a, b);
2333

```

4.3.4 进程退出钩子

进程退出钩子是目前 Node.js 官方文档中关于 C++ 原生扩展的最后一个例子。

钩子的名字叫作 `AtExit`，通过它注册进去的函数会在 Node.js 事件循环结束之后，JavaScript 运行时终止之前被调用；说白了就是，在 Node.js 程序马上要退出之前会调用一下通过它注册的函数。这略类似于 `process.on("exit", ...)`。

AtExit 函数在 node 命名空间下，即 `node::AtExit`。它的原型如下：

```
void node::AtExit(void (*cb)(void* arg), void* arg = 0);
```

- 第一个参数是一个函数，如下：

```
void Func(void* arg)
{
    // ...
}
```

- 第二个参数是需要传进第一个函数里面的参数，是 `void*` 指针型的。

通过 AtExit 注册进去的钩子函数将以栈¹的形式被存储；也就是说，在调用的时候，最后注册的函数最先被调用。

笔者将“19.at_exit_hook”中的代码转成流程图给大家看看，如图 4-2 所示。

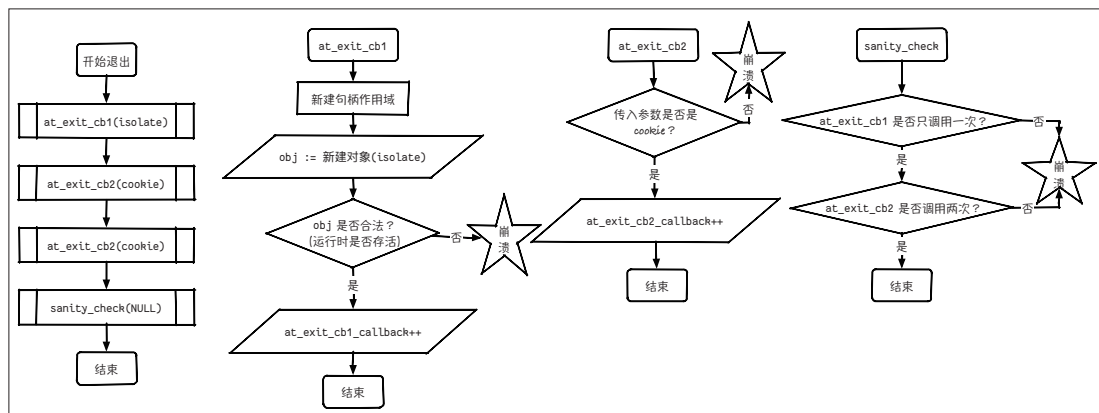


图 4-2 AtExit 钩子样例流程图

根据流程图我们知道，此样例共有 3 个钩子函数，分别是 `at_exit_cb1`、`at_exit_cb2` 和 `sanity_check`，而这 3 个钩子函数一共被注册了 4 次，其中 `at_exit_cb2` 被注册两次。于是钩子的调用链就是 `at_exit_cb1 → at_exit_cb2 → at_exit_cb2 → sanity_check`。照着这个流程图，我们先写一个架子出来。

```
static void at_exit_cb1(void* arg)
{
    ...
}
```

¹ 即先进后出的形式。


```

static void at_exit_cb2(void* arg)
{
    ...
}

static void sanity_check(void* arg)
{
    ...
}

void Init(Local<Object> exports)
{
    node::AtExit(sanity_check);
    node::AtExit(at_exit_cb2, cookie);
    node::AtExit(at_exit_cb2, cookie);
    node::AtExit(at_exit_cb1, exports->GetIsolate());
}

NODE_MODULE(addon, Init)

```

紧接着，尝试先实现 `at_exit_cb1` 的钩子函数。

```

static int at_exit_cb1_called = 0;
static void at_exit_cb1(void* arg)
{
    Isolate* isolate = static_cast<Isolate*>(arg);
    HandleScope scope(isolate);
    Local<Object> obj = Object::New(isolate);
    assert(!obj.IsEmpty()); // 断言 JavaScript 运行时是否还在运行
    assert(obj->IsObject());
    at_exit_cb1_called++;
}

```

然后是 `at_exit_cb2` 的钩子函数。

```

static char cookie[] = "yum yum";
static int at_exit_cb2_called = 0;
static void at_exit_cb2(void* arg)
{
    printf("2\n");
    assert(arg == static_cast<void*>(cookie));
    at_exit_cb2_called++;
}

```

最后，在 `sanity_check` 中判断一下 `at_exit_cb1_called` 和 `at_exit_cb2_called` 是否分别为 1 和 2 即可。

```
static void sanity_check(void*)
{
    assert(at_exit_cb1_called == 1);
    assert(at_exit_cb2_called == 2);
}
```

把上面的代码连起来就是“19. `at_exit_hook`”中 `at_exit.cc` 的代码了。为了方便大家追踪步骤，样例代码中每个函数的最开始都通过 `printf` 输出了一个相应的提示，好让大家知道哪个函数被执行了。

大家尝试着编译并运行“19. `at_exit_hook`”中的 `test.js` 文件吧。

```
$ node-gyp rebuild
...
$ node test.js
at_exit_cb1 called.
at_exit_cb2 called.
at_exit_cb2 called.
sanity_check called.
```

首先就是运行完毕后程序平安无事，并没有崩溃。其次就是运行的顺序如大家看到的提示一样依次执行。

想不想来一点儿刺激的？比如想一个办法让程序崩溃，以此来加深对 `AtExit` 的印象？比如我们看到了 `at_exit_cb2` 的判断是被调用两次，那么我们尝试着让它被调用 3 次看看效果？

```
void Init(Local<Object> exports)
{
    node::AtExit(sanity_check);
    node::AtExit(at_exit_cb2, cookie);
    node::AtExit(at_exit_cb2, cookie);
    node::AtExit(at_exit_cb2, cookie);
    node::AtExit(at_exit_cb1, exports->GetIsolate());
}
```

然后再编译和运行 `test.js` 的话，大家就会发现程序崩溃了——对于 `at_exit_cb2` 运行次数的断言失败。

```
$ node-gyp rebuild
...
```

```
$ node test.js
at_exit_cb1 called.
at_exit_cb2 called.
at_exit_cb2 called.
at_exit_cb2 called.
sanity_check called.
Assertion failed: (at_exit_cb2_called == 2), function sanity_check, file ../
at_exit.cc, line 36.
[1] 26624 abort node test.js
```

4.3.5 小结

本节主要介绍了如何封装一个 JavaScript 类，然后在类的对象中塞入一个与之绑定的 C++ 底层对象。在这个基础上，继续介绍了如何用封装对象工厂的形式去实例化类的对象，以及把这类对象传入 C++ 原生扩展函数时，如何把内置绑定的字段取出来使用。这些主要都依赖于 Node.js 封装的 `ObjectWrap` 类，其生命周期共生的原理在第 3 章中曾介绍过。

在类的这一基础概念介绍之后，在 4.3.4 节中还介绍了进程退出的钩子函数注册方法，是通过 `node::AtExit` 函数进行注册的。注册进去的函数是以栈的形式，在 Node.js 事件循环结束之后，JavaScript 运行时虚拟机退出之前被调用的。

4.3.6 参考资料

[1] Addons | Node.js v6.9.4 Documentation: <https://nodejs.org/docs/v6.9.4/api/addons.html>.

5

Node.js 原生抽象——NAN

NAN 的全称为 Native Abstractions for Node.js，其表现上是一个 Node.js 包。通过安装后，就得到了一堆 C++ 头文件，里面是一堆的宏。它主要为 Node.js 和 V8 跨版本提供了封装的宏，使得开发者不用关心各版本之间 API 的差异。

本章会对 NAN 进行介绍，使得开发者对使用 NAN 进行 Node.js 的 C++ 扩展开发方式有更熟练的开发姿势。

5.1 Node.js 原生模块开发方式的变迁

本节将从早期的 Node.js 开始，逐渐披露 Node.js 原生 C++ 扩展开发方式的变迁。

5.1.1 以不变应万变

虽然 Node.js 原生模块的开发方式有了很大的改变，但是有一些内容是不变的，至少到现在为止基本上没什么不兼容的变化。

1. 原生模块本质

正如笔者在 2.2.1 节中介绍的，Node.js 中的 C++ 模块实际上就是一个适配 Node.js 运行时的动态链接库，通过 JavaScript 代码中的 `process.dlopen()` 以及 C++ 中的 `DLOpen()` 函数对其进行引入。

从这一点来说，它一直没发生什么变化。

2. node-gyp

这个程序在 Node.js 中用于编译原生模块。在 4.1 节中也介绍过，其会搭配 `binding.gyp` 文件一起使用。

自从 Node.js v0.8 之后，它就跟 Node.js 黏上了。在此之前，Node.js 默认编译的帮助工具是 `node-waf`¹，老 Node.js 开发者对它应该不会陌生。

5.1.2 时代在召唤

第 *N* 套国际 Node.js 开发者原生 C++ 模块开发方式——时代在召唤。

除了前面介绍的一些不变的内容，还有很多内容是一直在变化的。虽说用老旧的开发方式也可以开发出能用的 C++ 原生模块，但是“旧不如新”。

而且，其实就目前来说 **node-gyp** 的地位也有可能在未来有所变化。因为当年 Chromium 也是通过 GYP 来管理其构建配置的，现如今已经步入了 GN² 的殿堂。这是否也意味着 `node-gyp` 某一天也会被一个也许叫作 `node-gn` 的东西取代呢？

话不多说，下面先来看看 Node.js 发展历程中的沧海桑田吧。

1. 黑暗时代——node-waf

在 Node.js 0.8 之前，通常在开发 C++ 原生模块的时候，是通过 `node-waf` 构建的。当然彼 `node-waf` 不是现在在 NPM 仓库上能搜到的 `node-waf`，当年那个 `node-waf` 早就“年久失修”了。

`node.waf` 使用一种叫 `wscript` 的文件来配置。自 Node.js 升级为 0.8 之后，就自带了 `node-gyp` 的支持，从此就不再需要 `wscript` 了。

不过因为那段时间各种使用 C++ 来开发的 Node.js 原生扩展的包为了兼容 0.8 前后版本的 Node.js，所以通常都是 `binding.gyp` 和 `wscript` 共存的。

大家可以看一下 `node-mysql-libmysqlclient` 这个包在早期的仓库文件³，它为了支持 `node-gyp`，有一个 `binding.gyp` 文件，且还存留着 `wscript` 配置文件。

1 年久失修，当前 NPM 上搜索到的 `node-waf` 已经不是当年的了，不过大家可以参考一下 Waf 的官方仓库：<https://github.com/waf-project/waf>。

2 https://chromium.googlesource.com/chromium/src/tools/gn/+HEAD/docs/quick_start.md

3 这里给出“9545ea7485fcc8b07b7c56c5ec3575938bfd4e5f”这个提交版本的仓库地址：<https://github.com/Sannis/node-mysql-libmysqlclient/tree/9545ea74>。

2. 封建时代——暴力！暴力！暴力！

node-waf 后来就被替换了，从此 Node.js 原生开发进入了封建时代。这个时候的 Node.js 原生 C++ 模块开发方式是非常暴力的，直接使用其提供的原生模块开发头文件，就跟本书之前章节介绍的那样。

开发者直接深入到 Node.js 的各种 API 及 Chrome V8 的 API。

下面举一个最简单的例子，在几年前，你的 Node.js C++ 原生扩展代码可能会是这样的：

```
Handle<Value> Echo(const Arguments& args)
{
    HandleScope scope;
    if(args.Length() < 1)
    {
        ThrowException(Exception::TypeError(
            String::New("Wrong number of arguments")));
        return scope.Close(Undefined());
    }

    return scope.Close(args[0]);
}

void Init(Handle<Object> exports)
{
    exports->Set(String::NewSymbol("echo"),
        FunctionTemplate::New(Echo)->GetFunction());
}
```

你是不是觉得一点都不熟悉？是不是想大叫一声——这不是我先前在本书中学到的姿势！这是 Node.js 在 0.10 版本时的 C++ 扩展代码样例。更多样例大家可以参考当时的 Node.js 文档¹。

这是一个最简单的 echo 函数，返回传进来的第一个参数。写作 JavaScript 相当于是这样的：

```
exports.echo = function() {
    if(arguments.length < 1) {
        throw new Error("Wrong number of arguments.");
    }
    return arguments[0];
}
```

¹ <https://nodejs.org/docs/v0.10.0/api/addons.html>

遗憾的是，这样的代码如果打成一个包，现在无论如何都无法安装，除非你用的是 0.10.x 的 Node.js 版本。

为什么这么说呢？因为这段代码的确是在 Node.js 0.10.x 的时候可以用。但是这段代码在高几个大版本的 Chrome V8 下就无法使用了。讲通俗一点就是没办法再编译通过了。

就拿本书中讲的 Node.js 6.x 版本的 Chrome V8 来说，函数声明对比是这样的：

```
Handle<Value> Echo(const Arguments& args);    // 0.10.x
void Echo(FunctionCallbackInfo<Value>& args); // 6.x
```

事实上，根本不需要等到 Node.js 6.x 版本。上面的代码到 Node.js 0.12 版本就已经无法再编译通过了。不只是函数声明的变化，就连句柄作用域的声明方式、各种 API 的变化也非常大。

如果要想它在 Node.js 6.x 下能编译，就需要改代码。读者根据本书先前讲述的内容，对它进行一个改装吧。就像这样：

```
void Echo(const FunctionCallbackInfo<Value>& args)
{
    Isolate* isolate = args.GetIsolate();
    if(args.Length() < 1)
    {
        isolate->ThrowException(Exception::TypeError(
            String::NewFromUtf8(isolate, "Wrong number of
arguments.")));
        return;
    }

    args.GetReturnValue().Set(args[0]);
}

void Init(Local<Object> exports)
{
    NODE_SET_METHOD(exports, "echo", Echo);
}
```

也就是说，以封建时代的方式进行 Node.js 原生模块开发的时候，一个版本的代码只能支持特定几个版本的 Node.js。一旦 Node.js 的底层 API 以及 Chrome V8 的 API 发生变化，而这些原生模块若依赖了那些有变化的 API，包就作废了（除非包的维护者支持新版的 API）。不过这样一来，旧版 Node.js 下就又无法编译通过新版的包了。

3. 城堡时代——Native Abstractions for Node.js

在经历了封建时代的尴尬局面之后，在 2013 年年中，一个“救世主”突然现世。

它的名字叫作 NAN¹，其全称为 Native Abstractions for Node.js，即 Node.js 原生模块抽象接口集。

NAN 由 Rod Vagg² 和 Benjamin Byholm³ 一手带大，寄名在 GitHub 的 Rod Vagg 账号下。并且在 Node.js 与 io.js 的“黑历史”⁴ 年代，这个在 GitHub 上的项目被移到了 io.js 的组织下；后来由于两家又重归于好，NAN 最终归属到了 nodejs 这个组织⁵ 下。

总之在 NAN 出现之后，Node.js 的原生开发方式进入了城堡时代，并且一直持续到现在，甚至可能会持续到好久之后。

说 NAN 是 Node.js 原生模块抽象接口集可能还是有点抽象，那么讲明白一点，它就是一堆宏判断。比如声明一个函数的时候，只需要通过下面的一个宏就可以了：

```
NAN_METHOD(Echo)
{
}
```

NAN 的宏会判断当前编译时的 Node.js 版本，根据不同版本的 Node.js 来展开不同的结果。这时就又会提到先前的两个函数声明对比了。

```
Handle<Value> Echo(const Arguments& args);    // 0.10.x
void Echo(FunctionCallbackInfo<Value>& args); // 6.x
```

在不同版本的 Node.js 中，NAN_METHOD 宏将会被 NAN 分别展开成上面的两行代码。当然 0.10.x 版本等的展开外面还有一层封装，这是为了能让开发者在写的时候能更方便。

而且 NAN 可不只是提供了 NAN_METHOD 一个宏，它还有一堆堆数不清的宏供开发者使用。

比如声明句柄作用域的 Nan::HandleScope、能以黑盒方式调用 libuv 进行事件循环上的异步操作的 Nan::AsyncWorker 等。

于是，在城堡时代，大家的 C++ 原生模块代码都像下面这样：

1 <https://github.com/nodejs/nan>

2 Node.js 核心贡献者之一，<https://github.com/rvagg/>。

3 NAN 核心贡献者之一，<https://github.com/kkooopa/>。

4 在这里表示人们对不愿提起的过往经历的称呼，并不是真正的黑暗历史。

5 <https://github.com/nodejs>


```

NAN_METHOD(Echo)
{
    if(info.Length() < 1)
    {
        Nan::ThrowError("Wrong number of arguments.");
        return info.GetReturnValue().Set(Nan::Undefined());
    }

    info.GetReturnValue().Set(info[0]);
}

NAN_MODULE_INIT(InitAll)
{
    Nan::Set(
        target,
        Nan::New<String>("echo").ToLocalChecked(),
        Nan::GetFunction(Nan::New<v8::FunctionTemplate>(Echo)).
        ToLocalChecked());
}

```

这样做的好处就是，代码只需要随着 NAN 的升级做改变就好，它会帮你兼容各不相同的 Node.js 版本，使其在任意版本都能被编译使用。

即使是 NAN 这样的库，也有自己的一个使命，而使命之外的东西会被逐渐剥离。比如 Node.js 0.10.x 和 Node.js 0.12.x 等版本就应该要退出历史舞台了，NAN 会逐渐放弃对它们的兼容和支持。

4. 帝国时代——符合 ABI 的 N-API

关于 N-API 的内容在本书第 9 章中会介绍。在这里作为 Node.js 原生模块开发方式变迁历史中的一个环节，也稍微提一下。

自从 2017 年 Node.js v8.0.0 发布之后，Node.js 就推出了全新的用于开发 C++ 原生模块的接口——N-API。

据官方文档所述，它的发音就是一个单独的 N，加上 API，即 4 个英文字母单独发音。

其与先前 3 个时代相比有什么不同呢？为什么会是更进一步的帝国时代呢？

首先，我们知道，即使是在 NAN 的开发方式下，一次编写好的代码在不同版本的 Node.js 下也需要重新编译；否则版本不符的话，Node.js 就无法正常载入一个 C++ 扩展。（代码只需要被开发者）一次编写，（就可以被使用者）到处编译。

而 N-API 与 NAPI 相比，它把 Node.js 的所有底层数据结构全部黑盒化，抽象成 N-API 中的接口。

不同版本的 Node.js 使用同样的接口，这些接口是稳定的、ABI 化的，即应用二进制接口¹（Application Binary Interface）。这使得在不同 Node.js 下，只要 ABI 的版本号一致，编译好的 C++ 扩展就可以直接使用，而不需要重新编译。事实上，在支持 N-API 接口的 Node.js 中，的确指定了当前 Node.js 所使用的 ABI 版本。

为了使得以后的 C++ 扩展开发、维护更方便，N-API 致力于以下几个目标：

- 以 C 的风格提供稳定的 ABI 接口。
- 消除 Node.js 版本的差异。
- 消除 JavaScript 引擎的差异（如 Chrome V8、Microsoft ChakraCore² 等）。

而这些 API 主要就是用来创建和操作 JavaScript 的值了，我们就再也不用直接使用 Chrome V8 提供的数据类型了。毕竟在 NAN 中，就算我们有时候看不到 Chrome V8 的影子，实际上在宏展开后仍有无数的 Chrome V8 数据结构。

为了达成上述隐藏的目标，N-API 的姿势就变成了这样：

- 提供头文件 `node_api.h`。
- 任何 N-API 调用都返回一个 `napi_status` 枚举，以表示这次调用成功与否。
- N-API 的返回值由于被 `napi_status` 占“坑”了，因此真实的返回值由传入的参数来继承，如传入一个指针让函数操作。
- 所有 JavaScript 数据类型都被黑盒类型 `napi_value` 封装，不再是类似于 `v8::Object`、`v8::Number` 等类型。
- 如果函数调用不成功，可以通过 `napi_get_last_error_info` 函数来获取最后一次出错的信息。

注意：哪怕是笔者在写作本节时的 Node.js v8.x 版本中，N-API 也仍处于一个试验状态。笔者个人认为其还有非常长的一段路要走，所以建议大家在生产环境中不要太过激进。当然，N-API 依然是大势所趋。不过对于使用旧版本的 Node.js 开发者来说，大家也不要着急。即使 N-API 是在 v8.x 才正式集成进 Node.js 的，在其他旧版本的 Node.js

¹ <https://zh.wikipedia.org/wiki/%E5%BA%94%E7%94%A8%E4%BA%8C%E8%BF%9B%E5%88%B6%E6%8E%A5%E5%8F%A3>

² 即微软查克拉引擎内核作为 JavaScript 引擎版本的 Node.js，欲了解具体内容可查看 <https://github.com/nodejs/node-chakracore>。

中依然可以将 N-API 作为外挂式的头文件¹使用，只不过无法做到跨版本的特性，这只是它做的向后兼容的一个事情而已。

有关 N-API 的一系列函数可以访问其文档以及阅读本书的第 9 章了解更多详情。

5.1.3 小结

本节讲述了 Node.js 原生 C++ 扩展开发方式的变迁史，之后介绍了 NAN 及其后续方式 N-API 的出现意义，使得读者对于 NAN 的出现意义有了一个大致的了解。

它的出现，主要是为了解决在不同版本的 Node.js 中，直接进行 C++ 扩展开发时由于接口不同而导致的代码不兼容的问题。它通过定义宏来封装不同的接口，对外提供了一致的接口。

5.1.4 参考资料

[1] 从暴力到 NAN 再到 N-API——Node.js 原生模块开发方式变迁：<http://gitbook.cn/m/mazi/article/593763494ec5fa29296acea0>.

[2] ABI 应用二进制接口：<http://www.cnblogs.com/liyonghelpme/archive/2010/06/10/4273556.html>.

5.2 基础开发

5.2.1 什么是 NAN

1. 存在的问题

NAN 的全称为 Native Abstraction for Node.js，即 Node.js 原生模块抽象接口集。

在本书的前面章节中，我们的所有内容都是基于 Node.js 6.x 进行解析的。不过在早期版本的 Node.js 中，笔者先前写的这些 C++ 原生扩展均无法正常编译通过。这会导致以下几个问题：

- 你写的包发布之后，别人在自己计算机上用其他不兼容的版本的 Node.js，就无法使用该包。
- 你写的包发布之后，在自己使用时，若你的服务器用的是不兼容的版本，也无法使用该包。

¹ 即 node-api 这个包，<https://github.com/nodejs/node-api>。

2. 问题的原因

那么，为什么我们的扩展无法运行在早期的 Node.js 版本中呢？为什么我们要担心自己写的扩展在未来有可能会失效呢？原因很简单，在 5.1.2 节中也曾提到，因为定义我们与 JavaScript 进行交互行为的并不是 Node.js，而是 Chrome V8，Node.js 只是我们使用的 JavaScript 代码与 Chrome V8 的一个胶水层。Chrome V8 的 API 随着时间的推移而改变，我们无法保证它会向下兼容，哪怕是未来的版本，我们也无法保证会兼容现在的版本。即使是 Chrome V8 开发人员可能会小心谨慎地对 API 做改变以保持它的稳定，但是在新版本中还是经常会出现一些或大或小的不兼容的变化。Node.js 会随着 Chrome V8 的迭代更新自身的 V8 依赖的版本。所以一旦 Node.js 的 Chrome V8 版本改变，我们的 C++ 原生插件就要跟着改变，并且改变后就无法兼容原来的 Node.js 版本了。

鉴于 Node.js 迭代快，在服务器上会存在各种新旧版本 Node.js 的历史包袱，比如有些服务器上的程序到现在还运行在 Node.js 0.10、0.12 以及 4.x 的版本中，所以把 Node.js 以及 Chrome V8 的一些 API 抽象出来就非常必要了，这也是 NAN 存在的意义。

3. NAN 的目标

NAN 为我们提供了比较稳定和干净的 API（以及宏），它就是对各版本 Node.js 和 Chrome V8 的 API 的一个封装，不同的 API 经过封装后对开发者暴露出统一、稳定的 API。

它由 Rod Vagg 开发，现在由 Node.js 团队管理，其聚焦两个目标：

- 为跨版本的 Node.js 提供稳定的 API。
- 提供一组实用的 API 来简化一些繁杂的开发流程，如异步开发。

值得一提的是，NAN 并不是高级的 API。因为它只有一小部分内容用于简化异步 C++ 原生扩展的开发。其主要目的还是简单地提供一些宏，用于替换一些原生的操作：

- V8 在 JavaScript 堆内存中创建元类型、对象。
- 定义导出函数。
- Node.js 的一些操作，如对象的封装。
- 其他操作。

不过，不要觉得自从有了 NAN 之后，读者先前学的关于 Chrome V8 的知识就全都没用了一——这些仍然适用，尤其是关于各种数据类型的操作、句柄和句柄作用域等概念。不同版本 Chrome V8 中变的只是一些细节的 API 而已。NAN 只是提供了它的一套 API，用于替换我们先前直接使用的 Chrome V8 API 而已。

5.2.2 安装和配置

1. NAN 安装

用 NAN 进行开发的话，首先要在你自己的 C++ 扩展项目或者包中加入它的依赖。包的名字就是 `nan`，直接通过 NPM 安装即可。

```
$ npm install --save nan
```

这样依赖，或多或少，NAN 已经存在于你的 `node_modules` 目录下或者相关目录下了，并且依赖也存在于你的 `package.json` 文件中。

2. binding.gyp 配置

在配置之前，我们先进入安装好的 NAN 目录下看看文件，主要文件里面有一堆的 C / C++ 头文件（*.h）、包文件（`package.json`）以及一个 JavaScript 文件 `include_dirs.js`，如图 5-1 所示。

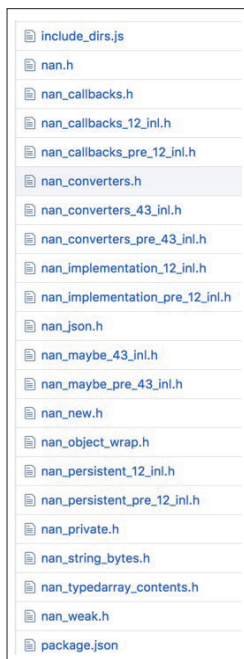


图 5-1 NAN 的主要文件

结合其目录下 `package.json` 中的 `main` 字段来看，当我们在 Node.js 源码中写 `require("nan")` 的时候，引入的就是 `include_dirs.js` 了。下面一起来看看，这个文件有什么用吧。

```
console.log(require('path').relative('.', __dirname));
```

看吧，它就输出了自身目录的绝对路径。可是这有什么用呢？

不知道大家是否还记得 `binding.gyp` 的这个配置文件，其中有一个字段是 `include_dirs`，代表了头文件的搜索路径，也就是能直接通过 `#include <>` 形式引入头文件的路径。我们把该路径写入 `include_dirs` 字段中，就能直接在自己的项目中通过 `#include <nan.h>` 来引入 NAN 了。

那么，要怎么配置 `binding.gyp` 呢？仔细回想一下它有“指令”一说，就是用 `<!` 等前缀表示一个指令。所以，在 `binding.gyp` 中我们就能通过这样的配置来引入 NAN 了。

```
"targets": [{
  ...
  "include_dirs": [
    "<!(node -e \"require('nan')\")"
  ]
},
...]
```

这样，在编译的时候就会是类似这样的编译指令了：

```
$ g++ ... -I<NAN 的路径> ...
```

5.2.3 先睹为快——搭上 NAN 的快车

在后面细讲 NAN 之前，先给大家“尝尝鲜”吧。为了使用 NAN，读者不需要再去包含 `node.h`，而直接包含 `nan.h` 即可。

注意，不管怎么变，模块注册的宏仍使用 `NODE_MODULE`，相信大家对它并不陌生。

1. 自动挡——看看代码

大家打开“20. nan echo”中的 `binding.gyp` 看看其是怎么配置的吧。

```
{
  "targets": [{
    "target_name": "echo",
    "sources": [ "echo.cc" ],
    "include_dirs": [
      "<!(node -e \"require('nan')\")"
    ]
  ]
}
```

```

    }}
}

```

一点儿也没错，就是使用了 GYP 中的指令特性，把 NAN 的路径引进来了。

然后接下来看看“20. nan echo/echo.cc”是怎么写的吧。

```

#include <nan.h>

namespace __nan_echo__ {

using v8::String;
using v8::FunctionTemplate;

NAN_METHOD(Echo)
{
    if(info.Length() < 1)
    {
        Nan::ThrowError("Wrong number of arguments.");
        return;
    }

    info.GetReturnValue().Set(info[0]);
}

NAN_MODULE_INIT(Init)
{
    Nan::Set(target, Nan::New<String>("echo").ToLocalChecked(),
        Nan::GetFunction(Nan::New<FunctionTemplate>(Echo)).
        ToLocalChecked());
}

NODE_MODULE(echo, Init)

}

```

2. 手动挡——讲讲展开

如果大家有兴趣，可以仔细看看本节中对之前的代码是如何展开的。不过如果你现在急于读完本书并进行开发的话，跳过本节的内容也是可以的。

首先要展开讲的是 `NAN_METHOD` 这个宏。它没有版本之分，无论什么版本的 Node.js 都是如下这样的声明：

```
namespace Nan {

typedef const FunctionCallbackInfo<v8::Value>& NAN_METHOD_ARGS_TYPE;
typedef void NAN_METHOD_RETURN_TYPE;

#define NAN_METHOD(name) \
    Nan::NAN_METHOD_RETURN_TYPE name(Nan::NAN_METHOD_ARGS_TYPE info)

}
```

也就是说，无论什么版本的 Node.js，之前那个 `Echo` 函数展开后会是这样的：

```
void Echo(const FunctionCallbackInfo<v8::Value>& info)
```

那么问题来了，明明低版本的 Node.js 函数声明不是这样的，为什么也被展开成这样呢？这事儿还得从 NAN 的目标是输出统一接口说起。如果它以类似于 5.1.2 节中封建时代所述的两种方式进行展开，低版本（Node.js 0.10.x）的函数返回值直接是一个句柄，而高版本的返回值是一个 `void`，那么在函数返回的时候就必然无法统一。

于是 NAN “曲线救国”，使得函数声明都展开成这样。由于低版本的 Node.js 没有 `FunctionCallbackInfo`，因此 NAN 就给它声明一个并封装起来。这个代码在 NAN 目录下的 `nan_callback_pre_12_inl.h` 中。

```
template<typename T>
class FunctionCallbackInfo {
    const v8::Arguments &args_;
    v8::Local<v8::Value> data_;
    ReturnValue<T> return_value_;
    v8::Persistent<T> retval_;

public:
    explicit inline FunctionCallbackInfo(
        const v8::Arguments &args
        , v8::Local<v8::Value> data) :
        args_(args)
        , data_(data)
        , return_value_(args.GetIsolate(), &retval_)
        , retval_(v8::Persistent<T>::New(v8::Undefined())) {}

    inline ~FunctionCallbackInfo() {
        retval_.Dispose();
        retval_.Clear();
    }
}
```



```

inline ReturnValue<T> GetReturnValue() const {
    return ReturnValue<T>(return_value_);
}

inline v8::Local<v8::Function> Callee() const { return args_.Callee(); }
inline v8::Local<v8::Value> Data() const { return data_; }
inline v8::Local<v8::Object> Holder() const { return args_.Holder(); }
inline bool IsConstructCall() const { return args_.IsConstructCall(); }
inline int Length() const { return args_.Length(); }
inline v8::Local<v8::Value> operator[](int i) const { return args_[i]; }
inline v8::Local<v8::Object> This() const { return args_.This(); }
inline v8::Isolate *GetIsolate() const { return args_.GetIsolate(); }

protected:
    static const int kHolderIndex = 0;
    static const int kIsolateIndex = 1;
    static const int kReturnValueDefaultValueIndex = 2;
    static const int kReturnValueIndex = 3;
    static const int kDataIndex = 4;
    static const int kCalleeIndex = 5;
    static const int kContextSaveIndex = 6;
    static const int kArgsLength = 7;

private:
    NAN_DISALLOW_ASSIGN_COPY_MOVE(FunctionCallbackInfo)
};

```

于是，在低版本的 Node.js 中使用 NAN，函数声明与高版本一致，只不过它还提供了一些假的数据结构供其使用。不止这么一个 `FunctionCallbackInfo`，它连 `ReturnValue`，也就是平时我们使用的 `info.GetReturnValue()` 的返回值也是伪造的。

不过还有一个问题，那就是在低版本中声明的函数 `Echo` 的返回值是 `void`，且参数是由 NAN 假设而成的，这样的一个函数定义是无法直接生成一个能供 JavaScript 调用的函数的。其实我们自己写的 `Echo` 也是经过一个 `FunctionCallbackWrapper` 封装的，代码如下：

```

typedef void(*FunctionCallback)(const FunctionCallbackInfo<v8::Value>&);

static
v8::Handle<v8::Value> FunctionCallbackWrapper(const v8::Arguments &args) {
    v8::Local<v8::Object> obj = args.Data().As<v8::Object>();
    FunctionCallback callback = reinterpret_cast<FunctionCallback>(
        reinterpret_cast<intptr_t>(

```

```

        obj->GetInternalField(kFunctionIndex).As<v8::External>()-
>Value());
    FunctionCallbackInfo<v8::Value>
        cbinfo(args, obj->GetInternalField(kDataIndex));
    callback(cbinfo);
    return ReturnValueImp<v8::Value>(cbinfo.GetReturnValue()).Value();
}

```

在通过 `Nan::New<FunctionTemplate>()` 时，它会向你返回一个经过其封装的对象，里面有一个内置字段就是你的函数的指针；在执行这个函数调用时，NAN 会先把开发者写的 `void Echo(...)` 函数取出来，并把事先准备好的伪造的 `FunctionCallbackInfo` 传入 `Echo` 函数运行；当代码执行经过开发者写的 `info.GetReturnValue()` 并结束 `Echo` 后，这个包装器会把开发者设置的内容取出来，直接返回给上一层。

这就是在笔者使用 `NAN_METHOD` 声明函数时，一定要使用 `Nan::New<Function>()` 来得到这个函数本体的原因了。因为它们是配套的，否则两个函数声明的方式会“驴唇不对马嘴”——无法配对。

这样的例子在 NAN 中非常常见。先通过定义一堆宏来统一 API；如果无法统一，就再定义一堆假的数据结构，经过封装后达到统一的效果。不止函数声明时是这样的，以后见到一些与新旧版 Node.js 的暴力开发方式相差甚远的代码时，你也不要觉得奇怪，大抵是 NAN 帮你做好了准备工作。

其他内容不再细述，有兴趣的读者可以自行去阅读 NAN 源码¹。

3. 踩油门——跑跑看

在介绍完 NAN 宏的样例展开后，读者可对事先写好的 `Echo` 源码进行测试。代码仍然在“20. nan echo”目录下。我们先使用 Node.js 0.10.x 进行编译，推荐大家使用 NVM² 来管理 Node.js 版本。

```

$ nvm use 0.10
...
$ node-gyp rebuild
...
$ node
> const addon = require("./build/Release/echo")
undefined

```

1 <https://github.com/nodejs/nan>

2 即 Node Version Manager，Node.js 版本管理器，用它可以方便地在机器上安装并维护多个 Node 的版本。可访问 <https://github.com/creationix/nvm> 获取更多内容。

```
> addon.echo(123123);
123123
> addon.echo("🐼");
'🐼'
> addon.echo("🐨");
'🐨'
```

然后再尝试用 4.x、6.x 以及 8.x 版本的 Node.js 进行测试。

```
$ nvm use 6
...
$ node
> const addon = require("./build/Release/echo");
Error: Module did not self-register.
    at Error (native)
    at Module.load (module.js:487:32)
    at tryModuleLoad (module.js:446:12)
    at Function.Module._load (module.js:438:3)
    at require (internal/module.js:20:19)
    at repl:1:15
    at sigintHandlersWrap (vm.js:73:12)
    at REPLServer.defaultEval (repl.js:346:29)
    at bound (domain.js:280:14)
    at REPLServer.<anonymous> (repl.js:545:10)
```

由于 Node.js 0.10 下编译的 C++ 模块与 Node.js 6.x 版本下编译的模块的模块注册函数不同，因此在 Node.js 6.x 版本下就找不到模块注册函数，于是直接引入在 0.10 版本的 Node.js 下编译的 C++ 模块就会暴露模块未进行自我注册的错误并崩溃。因此，读者要使用 node-gyp 重新编译一遍才能使用。

```
$ nvm use 6
...
$ node-gyp rebuild
...
$ node
> const addon = require("./build/Release/echo")
undefined
> addon.echo(123123);
123123
> addon.echo("🐼");
'🐼'
> addon.echo("🐨");
'🐨'
```

这回结果正常了。

5.2.4 基础帮助函数和宏

在介绍后续的宏之前，笔者先介绍 3 个基础的宏，以方便读者阅读后面的章节。

1. 模块初始化函数宏（`NAN_MODULE_INIT`）

在 Node.js 6.x 中，模块初始化函数如下面这样：

```
void Init(Local<Object> exports)
{
    // ...
}
```

或者是这样的：

```
void Init(Local<Object> exports, Local<Object> module)
{
    // ...
}
```

不过这些到了 NAN 中就变了，这个初始化函数的声明变成了一个宏，用于适配不同版本的 Node.js：

```
NAN_MODULE_INIT(Init)
{
    // ...
}
```

并且非常可惜的是，这个宏展开后，我们就只得到一个参数 `target`，它对应的是我们之前写的 `Init` 函数的第一个参数 `exports`。也就是说我们往 `target` 上挂东西，相当于往 `module.exports` 下面挂东西。

2. 模块函数导出（`Nan::Export()`）

`Nan::Export()` 主要用于模块初始化函数中，用于给 `target` 也就是以前的 `exports` 对象挂上 API 函数。

它的函数原型是这样的：

```
void Export(v8::Local<v8::Object> target, const char *name,
Nan::FunctionCallback f)
```

比如我们写了一个 `Echo` 函数，那么就能使用 `Nan::Export()` 进行导出了。

```

NAN_METHOD(Echo)
{
    // ...
}

NAN_MODULE_INIT(Init)
{
    Nan::Export(target, "echo", Echo);
}

```

与此同时，NAN 还为 `Nan::Export()` 函数提供了一个看起来更简单的宏 `NAN_EXPORT`，也就是说我们还能这么写：

```

NAN_METHOD(Echo)
{
    // ...
}

NAN_MODULE_INIT(Init)
{
    NAN_EXPORT(target, Echo);
}

```

导出的函数名与宏的第二个参数名相同，即这段代码导出到 JavaScript 中的函数名就是 `Echo`。不过这个宏不怎么推荐使用，因为根据 C++ 与 JavaScript 比较通用的代码规范来说，C++ 中函数声明推荐使用大驼峰¹的形式命名，而 JavaScript 中推荐使用小驼峰²的形式，这两种形式是有冲突的。比如上面代码中导出到 JavaScript 中的函数名要叫 `Echo`，是大驼峰的命名方法，并不适合 JavaScript。所以，还是推荐使用 `Nan::Export` 来根据不同的语言规范命名函数。

3. 回调函数调用 (`Nan::MakeCallback()`)

大家应该还记得，先前介绍 `Function` 这个数据类型的时候，提到过它有一个 `Call()` 的成员函数，用于调用这个函数得到返回值。不过其实 Node.js 提供了一个帮助函数供大家使用，那就是 `node::MakeCallback()`。我们来看看它在 Node.js 6.x 中的声明原型。

```

Local<Value> MakeCallback(
    Isolate* isolate,
    Local<Object> recv,

```

1 即帕斯卡 (Pascal) 命名法，变量名中每一个单词的首字母都采用大写字母，例如：`FirstName`、`LastName`、`CamelCase`。

2 与大驼峰命名法相似，只不过第一个单词以小写字母开始，其余单词的首字母大写。例如：`firstName`、`lastName`。

```
Local<Function> callback,
int argc,
Local<Value>* argv);
```

针对不同的场景，这个 `MakeCallback()` 函数提供了不同的重载，不过我们最关心的还是书中列出来的这一个。从结果来看，这个函数与 `Function::Call` 差不多。但是从语义上来讲，这个函数更偏向用于回调函数的调用，这样写代码的时候让维护者或者贡献者能明白这个调用是一个回调函数的调用。

为了版本之间的 `MakeCallback` 能互相兼容，NAN 也提供了一个 `Nan::MakeCallback` 函数，供大家使用。同样地，我们最关心的也是类似于上面的这么一个重载。

```
v8::Local<v8::Value> Nan::MakeCallback(v8::Local<v8::Object> target,
                                       v8::Local<v8::Function> func,
                                       int argc,
                                       v8::Local<v8::Value>* argv);
```

4. 用实例说话

让我们结合前面讲的模块初始化函数宏、模块函数导出和本节的回调函数调用内容，修改一下之前随书代码中“11. array prototype map”的样例，将其改造成 NAN 适用的代码吧。改造后的代码在“21. nan array prototype map”中。下面打开来看看吧。

```
void Map(const FunctionCallbackInfo<Value>& args)
{
    Local<Array> array = args[0].As<Array>();
    Local<Function> func = args[1].As<Function>();

    Local<Array> ret = Nan::New<Array>(array->Length());

    Local<Value> null = Nan::Null();
    Local<Value> a[3] = { Nan::New<Object>(), null, array };
    for(uint32_t i = 0; i < array->Length(); i++)
    {
        a[0] = array->Get(i);
        a[1] = Nan::New<Int32>(i);

        Local<Value> v = Nan::MakeCallback(args.This(), func, 3, a);
        ret->Set(i, v);
    }

    args.GetReturnValue().Set(ret);
}
```

```
NAN_MODULE_INIT(Init)
{
    Nan::Export(target, "map", Map);
}

NODE_MODULE(map, Init)
```

从上述代码中能看到，我们把模块初始化函数改用了 `NAN_MODULE_INIT` 宏来声明，函数挂载用 `Nan::Export` 代替。在 `Map` 函数中，使用 `Nan::MakeCallback` 对函数进行调用。这么一来，3 个知识点都过了一遍。

最后，随意使用一个版本的 Node.js 对其进行编译和运行吧。

```
$ node-gyp rebuild
...
$ node
> const addon = require("../build/Release/map");
undefined
> addon.map([1,2], function() { return arguments; });
[ { '0': 1,
    '1': 0,
    '2': [ 1, 2 ] },
  { '0': 2,
    '1': 1,
    '2': [ 1, 2 ] } ]
```

5.2.5 忽略 node_modules

通过前面的介绍，我们明白了不同版本 Node.js 下的 C++ 扩展是无法互相使用的，必须重新编译。

在众多的“不要把 `node_modules` 目录加到我们的版本库中”的理由又多了一条——你把自己这里编译好的 C++ 扩展模块依赖加到版本库中，就会导致别人无法运行你的项目，他迟早还是要重新安装一遍依赖的。

5.2.6 小结

本节介绍了原生暴力开发 C++ 模块时会遇到的问题，如代码无法跨特定版本进行编译，并基于此给出了 NAN 存在的意义——就是为了让代码能跨版本。

后续又给出了 NAN 的安装方式和配置方法，使其能集成进我们自己的 C++ 扩展当中。在最后，给出了一个 `Echo` 的样例项目，让大家初步知道使用 NAN 进行开发的姿势与原始的姿势有何不同，并且通过样例讲解了它是如何做到跨版本机制的——使用宏包装以及假数据结构的适配。

在一切尘埃落定的时候，运行了样例、给出了效果，并且尝试着不重新编译时跨版本使用 Node.js，暴露出了在不同版本下 Node.js 的 C++ 原生扩展不能互用的问题。

5.2.7 参考资料

[1] Scott Frees. C++ and Node.js Integration[EB/OL]. <https://scottfrees.com/ebooks/nodecpp/>.

5.3 JavaScript 函数

笔者在前文中讲过，在 Chrome V8 层要根据自己写的函数封装成一个 JavaScript 函数的话，需要通过 `FunctionTemplate` 来包装。为此，读者自己实现的函数必须要按照某种规范来，如在 Node.js v6.x 中会如下所示：

```
void Func(const FunctionCallbackInfo<Value>& info);
```

NAN 为这些函数的声明提供了一堆实用的帮助函数，供不同版本间的 Node.js 统一使用。

注意：本节内容将会对照 NAN 文档的 JavaScript 可访问函数相关内容进行讲解。

5.3.1 函数参数类型

在 5.2 节笔者提到了，NAN 为旧版定义了一堆伪造的兼容新版开发的数据结构。在后文会逐一给出其类的声明原型。不过其原型代码可能会随着时间的推移在 NAN 中不断迭代。读者可以通过阅读本节内容了解一个大概，然后去往 NAN 的代码仓库地址获取一个最新的内容。

1. 普通函数参数类型 (`Nan::FunctionCallbackInfo`)

新版的 Node.js 无可厚非，函数参数的类型是 `FunctionCallbackInfo`，但是旧版的 Node.js 可不是这样的。所以 NAN 为旧版的 Node.js 定义了一个 `FunctionCallbackInfo`。NAN 为旧版 Node.js 定义的 `FunctionCallbackInfo` 可能是这样的：

```
template<typename T> class FunctionCallbackInfo {
public:
```



```

ReturnValue<T> GetReturnValue() const;
v8::Local<v8::Function> Callee();
v8::Local<v8::Value> Data();
v8::Local<v8::Object> Holder();
bool IsConstructCall();
int Length() const;
v8::Local<v8::Value> operator[](int i) const;
v8::Local<v8::Object> This() const;
v8::Isolate *GetIsolate() const;
};

```

暴露出来的 API 跟新版 V8 的 `FunctionCallbackInfo` 一模一样。不过为了让开发者在开发的时候，根本不需要关心 Node.js 版本，在实际开发中，哪怕是新版的 Node.js，它也不再是 `v8::FunctionCallbackInfo` 了，而是使用 `Nan::FunctionCallbackInfo` 代替。

这样一来，我们在自己声明的函数中，就跟在新版本 Node.js 中写 C++ 扩展的时候写法一致，都是通过 `info.GetReturnValue().Set()` 来返回值，并且能通过 `info.Holder()` 来获取调用者等。

2. 属性访问函数参数类型 (`Nan::PropertyCallbackInfo`)

与普通函数不同，属性访问函数主要是那些访问器、拦截器函数。关于访问器和拦截器笔者在 3.6.3 节中也曾讲到，所以大家对 `PropertyCallbackInfo` 这个类型应该不陌生。

有一点与普通函数一样的是，NAN 也为低版本的属性访问函数虚构了一个参数的类型，叫 `Nan::PropertyCallbackInfo`，而新版的 `Nan::PropertyCallbackInfo` 实际上就是 `v8::PropertyCallbackInfo` 的一个映射而已。

下面来看一下旧版 Node.js 中，NAN 对于 `PropertyCallbackInfo` 的一个声明原型。

```

template<typename T> class PropertyCallbackInfo : public
PropertyCallbackInfoBase<T> {
public:
    ReturnValue<T> GetReturnValue() const;
    v8::Isolate* GetIsolate() const;
    v8::Local<v8::Value> Data() const;
    v8::Local<v8::Object> This() const;
    v8::Local<v8::Object> Holder() const;
};

```

3. 函数返回类型 (Nan::ReturnValue)

在旧版的 Node.js 中，是没有 ReturnValue 一说的，它的函数会通过直接返回一个句柄来返回一个数据。所以 NAN 对旧版的函数进行了一层封装，在封装的函数中调用开发者写的函数，并传入一个自己伪造的 ReturnValue 供其返回。

这样我们在自己的 Node.js C++ 扩展中仍然能以新版 Node.js 的开发方式进行开发，就像下面这样：

```
void EmptyArray(const Nan::FunctionCallbackInfo<v8::Value>& info) {
    info.GetReturnValue().Set(Nan::New<v8::Array>());
}
```

笔者在这里给出 NAN 对于 ReturnValue 的一个声明。

```
template<typename T> class ReturnValue {
public:
    // 句柄设置
    template <typename S> void Set(const v8::Local<S> &handle);
    template <typename S> void Set(const Nan::Global<S> &handle);

    // 快速元数据设置
    void Set(bool value);
    void Set(double i);
    void Set(int32_t i);
    void Set(uint32_t i);

    // 快速 JavaScript 元数据设置
    void SetNull();
    void SetUndefined();
    void SetEmptyString();

    // Isolate 对象的便捷获取
    v8::Isolate *GetIsolate() const;
};
```

5.3.2 函数声明

通过 5.3.1 节的讲解我们明白了，在使用 NAN 写扩展的时候，函数声明的写法基本遵从了新版 Node.js 的写法，旧版会被包装成与新版兼容。

1. 普通函数声明

普通函数声明的函数类型是 `FunctionCallback`，它是这样被定义的。

```
typedef void(*FunctionCallback)(const FunctionCallbackInfo<v8::Value>&);
```

不过在使用了 NAN 之后，`FunctionCallback` 就应该加上 `Nan` 这个命名空间前缀了，即 `Nan::FunctionCallback`。

大家再回想一下 5.2.3 节介绍的 `Echo` 函数吧，实际上它的声明就是这样的：

```
void Echo(const Nan::FunctionCallbackInfo<v8::Value>& info)
```

并且，你不需要在其中声明一个句柄作用域 `HandleScope`。在新版中，我们本身就不需要创建它；而在旧版 Node.js 中，NAN 已经为你在外层包装好了作用域。实际上新旧版本的句柄作用域创建方法也是不一样的。

既然大家知道了 NAN 中普通函数声明与 Node.js 6.x 无异，就应该很容易理解这么一段样例代码了。

```
// .h:
class Foo : public Nan::ObjectWrap {
    ...

    static void Bar(const Nan::FunctionCallbackInfo<v8::Value>& info);
    static void Baz(const Nan::FunctionCallbackInfo<v8::Value>& info);
}

// .cc:
void Foo::Bar(const Nan::FunctionCallbackInfo<v8::Value>& info) {
    ...
}

void Foo::Baz(const Nan::FunctionCallbackInfo<v8::Value>& info) {
    ...
}
```

除了普通函数声明的姿势不变之外，其实 NAN 还为普通函数声明提供了一个简化的宏。这个宏在先前也介绍过了，就是 **NAN_METHOD(函数名)**。在这个宏中，参数的名字是一个固定值 `info`，所以平时我们均用 `info.<成员名>` 来进行成员的调用，如 `info.Holder()`。

于是上面的这段代码就可以被改成下面这样：

```
// .h:
class Foo : public Nan::ObjectWrap {
    ...

    static NAN_METHOD(Bar);
    static NAN_METHOD(Baz);
}

// .cc:
NAN_METHOD(Foo::Bar) {
    ...
}

NAN_METHOD(Foo::Baz) {
    ...
}
```

实际上我们无法保证 Node.js 和 Chrome V8 的 API 的变更是如何的，如哪里一定会变，或者哪里一定不会变。所以，也许 FunctionCallback 的形式在某天会失效。为了能做好兼容，这里还是推荐使用 NAN_METHOD 这种黑盒的形式来声明你的函数。后续章节中提到的一些相关处，能使用 NAN 封装的帮助 API（helper API）的地方就尽量使用 NAN，尽量少用原生的 API。

2. 访问器声明

访问器相当于 JavaScript 中的 Getter 和 Setter，这一点笔者在 3.6.3 节中也曾提到过。

实际上在 NAN 中，访问器的声明跟 Node.js 6.x 一样，把 PropertyCallbackInfo 的命名空间改为 Nan 即可。

下面列一下 Nan::GetterCallback 和 Nan::SetterCallback 的定义。

```
namespace Nan {

typedef void(*GetterCallback) (v8::Local<v8::String>,
                              const Nan::PropertyCallbackInfo<v8::Value>&);

typedef void(*SetterCallback) (v8::Local<v8::String>,
                              v8::Local<v8::Value>,
                              const Nan::PropertyCallbackInfo<void>&);

}
```

这两个函数类型定义跟先前讲的访问器声明一样，实际上用起来也差不多：

```
void GetterName(v8::Local<v8::String> property,
               const Nan::PropertyCallbackInfo<v8::Value>& info) {
    ...
}

void SetterName(v8::Local<v8::String> property,
               v8::Local<v8::Value> value,
               const Nan::PropertyCallbackInfo<void>& info) {
    ...
}
```

在访问器中，无论是新版 Node.js 还是旧版 Node.js，都不需要在里面声明句柄作用域，原因跟前文中所述相同。

可能大家猜到了笔者接下来要说什么了，还是两个宏。与普通函数声明的帮助宏对应，笔者接下来会给出访问器中 Getter 和 Setter 声明的帮助宏。

```
NAN_GETTER(Getter 名)
{
    // 这里是 Getter 代码
}

NAN_SETTER(Setter 名)
{
    // 这里是 Setter 代码
}
```

通过访问器的帮助宏声明的函数中，参数名是固定的：

- 传入的属性名的参数名字就是 property；
- PropertyCallbackInfo 的参数名是固定的 info；
- 而 Setter 中的设置值参数名是固定的 value。

大家使用的时候不要用错了。

3. 拦截器声明

拦截器是笔者在 3.6.3 节中讲的关于 Chrome V8 的另一个概念，根据传入的属性名来返回相应的内容。

在 NAN 中，拦截器的函数声明也与 Node.js v6.x 一致。拦截器中有 5 种类型的函数，分别是 Getter、Setter、Enumerator、Deleter 和 Query。下面我们再温习一下它们的定义吧，这里给出它们在 Nan 命名空间中的样式。

```
namespace Nan {

// 映射型拦截器

typedef void(*PropertyGetterCallback) (v8::Local<v8::String>,
                                       const Nan::PropertyCallbackInfo<v8::Value>&);

typedef void(*PropertySetterCallback) (v8::Local<v8::String>,
                                       v8::Local<v8::Value>,
                                       const Nan::PropertyCallbackInfo<v8::Value>&);

typedef void(*PropertyEnumeratorCallback) (const Nan::PropertyCallbackInfo<v8::Array>&);

typedef void(*PropertyDeleterCallback) (v8::Local<v8::String>,
                                       const Nan::PropertyCallbackInfo<v8::Boolean>&);

typedef void(*PropertyQueryCallback) (v8::Local<v8::String>,
                                       const Nan::PropertyCallbackInfo<v8::Integer>&);

// 索引型拦截器

typedef void(*IndexGetterCallback) (uint32_t,
                                    const PropertyCallbackInfo<v8::Value>&);

typedef void(*IndexSetterCallback) (uint32_t,
                                    v8::Local<v8::Value>,
                                    const PropertyCallbackInfo<v8::Value>&);

typedef void(*IndexEnumeratorCallback) (const PropertyCallbackInfo<v8::Array>&);

typedef void(*IndexDeleterCallback) (uint32_t,
                                    const PropertyCallbackInfo<v8::Boolean>&);

typedef void(*IndexQueryCallback) (uint32_t,
```

```
const PropertyCallbackInfo<v8::Integer>&);  
}
```

不用笔者多说什么了，其实这一大堆函数声明也是有宏的。下面直接给大家看看宏名吧。

```
// 映射型拦截器  
NAN_PROPERTY_GETTER(Getter 名)  
NAN_PROPERTY_SETTER(Setter 名)  
NAN_PROPERTY_ENUMERATOR(Enumerator 名)  
NAN_PROPERTY_DELETER(Deleter 名)  
NAN_PROPERTY_QUERY(Query 名)  
  
// 索引型拦截器  
NAN_INDEX_GETTER(Getter 名)  
NAN_INDEX_SETTER(Setter 名)  
NAN_INDEX_ENUMERATOR(Enumerator 名)  
NAN_INDEX_DELETER(Deleter 名)  
NAN_INDEX_QUERY(Query 名)
```

最后还是大家最关心的，每个宏展开后，参数名各是什么。由于函数太多，归类略麻烦，因此这里各给出一个样例供大家参考，参数名就是各样例所示的那样。

```
// 映射型拦截器  
void PropertyGetterName(Local<String> property, const  
PropertyCallbackInfo<Value>& info)  
void PropertySetterName(Local<String> property, Local<Value> value, const  
PropertyCallbackInfo<Value>& info)  
void PropertyEnumeratorName(const PropertyCallbackInfo<Array>& info)  
void PropertyDeleterName(Local<String> property, const  
PropertyCallbackInfo<Boolean>& info);  
void PropertyQueryName(Local<String> property, const  
PropertyCallbackInfo<Integer>& info);  
  
// 索引型拦截器  
void IndexGetterName(uint32_t index, const PropertyCallbackInfo<Value>&  
info)  
void IndexSetterName(uint32_t index, Local<Value> value, const  
PropertyCallbackInfo<Value>& info)  
void IndexEnumeratorName(const PropertyCallbackInfo<Array>& info)  
void IndexDeleterName(uint32_t index, const PropertyCallbackInfo<Boolean>&  
info);  
void IndexQueryName(uint32_t index, const PropertyCallbackInfo<Integer>&  
info);
```

也就是说，如果我们声明一个索引型拦截器的 Getter，它可能像下面这样：

```
NAN_PROPERTY_GETTER(Send)
{
    if(!property->IsString())
    {
        return;
    }

    ...

    MaybeLocal<Value> ret = Nan::Call(func, info.Holder(), 1, argv);
    info.GetReturnValue().Set(ret.ToLocalChecked());
}
```

5.3.3 函数设置

函数光声明有什么用呢？自从函数声明之后，却没有地方把它设置起来，你是否也升起了一股“要你何用”的感慨呢？如果使用的是新版的 Node.js，则没什么疑问，由于 NAN 的宏展开与自身的开发方式差异不大，因此还是能直接用老旧的知识去解决的。

本节介绍如何将这些通过 NAN 的各种宏定义的函数设置到我们的扩展中，并暴露给外界使用。

1. Nan::New<FunctionTemplate>

这里提前讲一点，那就是读者能通过 `Nan::New<FunctionTemplate>` 这个模板函数，传入读者声明的普通 JavaScript 函数指针，就能得到一个函数模板了——无论版本新旧。

```
NAN_METHOD(Echo)
{
    ...
}

某个函数 ()
{
    Local<FunctionTemplate> tpl = Nan::New<FunctionTemplate>(Echo);
    Local<Function> func = tpl->GetFunction();
}
```

这样一来，读者就能得到心仪的 `FunctionTemplate` 甚至是 `Function` 的对象了。这个方法希望大家牢记于心，它是 `Nan::New` 这个模板家族的一个用法。

2. 普通函数设置

除了按部就班，通过 `Nan::New<FunctionTemplate>` 获得一个函数对象之外，NAN 还提供了一些便利的函数供大家使用。比如读者能通过 `Nan::SetMethod` 来直接给某个对象或者模板（无论是对象模板还是函数模板）下面挂上某个属性名的函数。

细心的读者可能发现了，上面说的是某个对象或者模板，也就是说这个函数一共有两个重载，如下：

```
void Nan::SetMethod(v8::Local<v8::Object> recv,
                   const char *name,
                   Nan::FunctionCallback callback)
void Nan::SetMethod(v8::Local<v8::Template> templ,
                   const char *name,
                   Nan::FunctionCallback callback)
```

那么，读者就能在扩展的 `Init` 函数中这么写：

```
NAN_METHOD(Echo)
{
    if(info.Length < 1) return Nan::ThrowError("...");
    info.GetReturnValue().Set(info[0]);
}

void Init(Local<Object> target)
{
    Nan::SetMethod(target, "echo", Echo);
}
```

在 NAN 的帮助宏中，模块初始化函数的宏中参数名就是 `target`，而不是读者先前熟知的 `module` 或者 `exports` 了。所以，这里举例的时候，笔者直接用上了 `target`。

`Nan::SetMethod` 的效果与 `Nan::Export` 差不多，不过笔者在 5.2.4 节中说过了，在我们的日常开发中不推荐使用 `Nan::Export` 作为导出的帮助函数，所以平时还是推荐使用 `Nan::SetMethod`。并且它不只是可以在初始化的时候使用，在任何你需要对对象挂一个函数的时候都可以使用。

3. 对象原型链设置

关于原型链，我们很清楚在 Node.js 6.x 中，它是在 `FunctionTemplate::PrototypeTemplate` 中挂载属性的。这当然在 NAN 中也给了便利的函数，那就是 `Nan::SetPrototypeTemplate`。

```
void Nan::SetPrototypeTemplate(v8::Local<v8::FunctionTemplate> templ,
                              const char *name,
                              v8::Local<v8::Data> value);
```

使用的姿势是这样的：

```
NAN_METHOD(Echo)
{
    // ...
}

void Init(Local<Object> target)
{
    Local<FunctionTemplate> tpl = Nan::New<FunctionTemplate>();
    Nan::SetPrototypeTemplate(tpl, "echo", Echo);
}
```

知道了这些之后，我们一起来改造一下先前讲的“10. function template inherit”吧，顺便把前面的普通函数设置以及 `Nan::New<FunctionTemplate>` 都过一遍。

着手打开“22. nan set method”中的 `prototype/addon.cc` 源码，逐个函数进行解析。

```
NAN_METHOD(SetName)
{
    Nan::Set(info.This(), Nan::New("name").ToLocalChecked(), info[0]);
}

NAN_METHOD(Summary)
{
    Local<Object> self = info.This();
    char temp[512];

    Nan::Utf8String type(
        Nan::Get(self, Nan::New("type")
            .ToLocalChecked()).ToLocalChecked()->ToString());

    Nan::Utf8String name(
        Nan::Get(self, Nan::New("name")
            .ToLocalChecked()).ToLocalChecked()->ToString());

    snprintf(temp, 511, "%s is a/an %s.", *name, *type);
    info.GetReturnValue().Set(Nan::New(temp).ToLocalChecked());
}
```

以上代码就是 `Pet` 原型链上面的函数改造，其已经有非常浓重的 NAN 影子了。例如 `NAN_METHOD` 等，以及 `Nan::Set` 和 `Nan::Get` 的作用类似于 Chrome V8 的对象属性设置和获取，`Nan::New` 是新建一个 Chrome V8 的数据类型，所有这些在后续章节都会被提到。

有了两个原型链上的函数之后，我们就要实现构造函数了，并且将它的原型链设置好。

```
NAN_METHOD(Pet)
{
    Local<Object> self = info.This();

    Nan::Set(
        self,
        Nan::New("name").ToLocalChecked(),
        Nan::New("Unknown").ToLocalChecked());

    Nan::Set(
        self,
        Nan::New("type").ToLocalChecked(),
        Nan::New("animal").ToLocalChecked());

    info.GetReturnValue().Set(self);
}

NAN_MODULE_INIT(Init)
{
    Local<FunctionTemplate> pet = Nan::New<FunctionTemplate>(Pet);
    Nan::SetPrototypeMethod(pet, "setName", SetName);
    Nan::SetPrototypeMethod(pet, "summary", Summary);

    Local<Function> pet_cons = pet->GetFunction();

    ...

    Nan::Set(target, Nan::New("Pet").ToLocalChecked(), pet_cons);

    ...
}
```

这段代码中就有不少这几节内容中提到的知识点了，`NAN_MODULE_INIT` 函数中的第一句代码就是通过 `Nan::New` 来声明一个基于 `Pet` 的函数模板，并通过 `Nan::SetPrototypeMethod()` 来设置原型链函数。之后通过 `pet->GetFunction()` 获得函数句柄，再用 `Nan::Set` 把它挂载到 `target`（也就是以前说的 `exports` 对象）下面。

接下去就是继承这块了，有了这些基础后，继承的改造也就异常简单了。

```

NAN_METHOD(Dog)
{
    Local<Object> self = info.This();
    Local<Function> super = Nan::New(cons);

    Nan::Call(super, self, 0, NULL);
    Nan::Set(
        self,
        Nan::New("type").ToLocalChecked(),
        Nan::New("dog").ToLocalChecked());

    info.GetReturnValue().Set(self);
}

NAN_MODULE_INIT(Init)
{
    ...

    Local<Function> pet_cons = pet->GetFunction();

    cons.Reset(pet_cons);

    Local<FunctionTemplate> dog = Nan::New<FunctionTemplate>(Dog);
    dog->Inherit(pet);

    Local<Function> dog_cons = dog->GetFunction();

    ...
    Nan::Set(target, Nan::New("Dog").ToLocalChecked(), dog_cons);
}

```

在 Dog 构造函数中，通过 `Nan::New` 从持久句柄中获取到其对应的本地句柄也就是 Pet 构造函数。然后通过 `Nan::Call(..., self, ...)` 去调用这个函数以继承。接下来还是俗套的 `Nan::Set` 以及返回值了。

不过这里值得注意的一点就是，笔者用了 `Nan::Call` 而不是 `Nan::MakeCallback`，是因为当前调用 `super` 的语义是为了继承，并不是说它是一个回调函数，所以为了语义明晰，还是选择了 `Nan::Call` 来调用父类的构造函数。

最后，跑到其他任意版本的 Node.js 下编译运行看看吧：

```

$ nvm use 0.10
...
$ node-gyp rebuild

```

```

...
$ node
> var addon = require("./build/Release/prototype");
undefined
> var pet = new addon.Pet();
undefined
> pet.summary();
'Unknown is a/an animal.'
> var dog = new addon.Dog();
undefined
> dog.setName("蛋花汤");
undefined
> dog.summary();
'蛋花汤 is a/an dog.'

```

4. 访问器设置

既然前面介绍了 NAN 中如何声明访问器函数，那么肯定就能把它设置到某个对象模板上。在 NAN 里面，访问器设置是通过 `Nan::SetAccessor()` 进行的。

这是我们常用的一个访问器设置函数的重载：

```

bool SetAccessor(v8::Local<v8::Object> obj,
                v8::Local<v8::String> name,
                Nan::GetterCallback getter,
                Nan::SetterCallback setter = 0,
                v8::Local<v8::Value> data = v8::Local<v8::Value>(),
                v8::AccessControl settings = v8::DEFAULT,
                v8::PropertyAttribute attribute = v8::None)

```

每个参数的意思对照着先前介绍的 Chrome V8 原生访问器设置函数阅读即可。

大家应该也猜到了，对于访问器设置来说，我们也是有先前的代码可以改造的。现在就可以把“5. object template accessor”中的源码改造成“22. nan set method”中的 `accessor/addon.cc` 了。由于篇幅所限，因此笔者只改造全局变量那段访问器的代码。

```

int x = 0;

NAN_GETTER(Getter)
{
    info.GetReturnValue().Set(x);
}

NAN_SETTER(Setter)

```

```

{
    x = value->Int32Value();
}

NAN_MODULE_INIT(Init)
{
    Local<ObjectTemplate> tpl = Nan::New<ObjectTemplate>();
    Nan::SetAccessor(tpl, Nan::New("x").ToLocalChecked(), Getter, Setter);

    Local<Object> ret = ((Nan::MaybeLocal<Object>)tpl->NewInstance()).
    ToLocalChecked();
    Nan::Set(target, Nan::New("obj").ToLocalChecked(), ret);
}

```

很简单，无非是把各种原生的函数声明变成了 NAN 版本的，然后把 `tpl->SetAccessor` 变成了 `Nan::SetAccessor` 而已。

下面来运行一下验证吧。

```

$ nvm use 0.10
...
$ node-gyp rebuild
...
$ node
> var addon = require("../build/Release/accessor").obj;
undefined
> addon
{ x: 0 }
> addon.x = 233;
233
> addon
{ x: 233 }

```

5. 拦截器设置

拦截器的设置变动也与访问器设置的变动差不多，由某个模板的成员函数变成了 `Nan` 的函数了。

```

void SetNamedPropertyHandler(v8::Local<v8::ObjectTemplate> tpl,
                             Nan::PropertyGetterCallback getter,
                             Nan::PropertySetterCallback setter = 0,
                             Nan::PropertyQueryCallback query = 0,
                             Nan::PropertyDeleterCallback deleter = 0,
                             Nan::PropertyEnumeratorCallback enumerator = 0,
                             v8::Local<v8::Value> data =
v8::Local<v8::Value>())

```

```
void SetIndexedPropertyHandler(v8::Local<v8::ObjectTemplate> tpl,
                              Nan::IndexGetterCallback getter,
                              Nan::IndexSetterCallback setter = 0,
                              Nan::IndexQueryCallback query = 0,
                              Nan::IndexDeleterCallback deleter = 0,
                              Nan::IndexEnumeratorCallback enumerator = 0,
                              v8::Local<v8::Value> data =
v8::Local<v8::Value>())
```

在这里值得注意的一点是，由于早期的 Node.js 并没有类似于 `kOnlyInterceptStrings` 之类的 `PropertyHandlerFlags` 概念，因此在统一的 NAN 设置拦截器函数中，也没有这个参数，在高版本中只能用它的默认值。如果你有其他的 `PropertyHandlerFlags` 需求，那只能抛弃对旧版 Node.js 的兼容了。

由于篇幅的原因，本节的样例代码改造就不放出来了，请大家自行想象应当如何改造“6. mapped property interceptor”这个项目。大家在思考完成之后，可以参考“22. nan set method”中的 `interceptor/addon.cc` 源码文件。

在这里给出一个小提示，由于 `Nan::SetNamedPropertyHandler` 函数没有了 `PropertyHandlerFlags` 这个设置，因此我们就不能像改造前一样使用 `kOnlyInterceptStrings` 了。也就是说，在新版的 Node.js 中，一个 ECMAScript 6 中的 `Symbol` 也有可能被传入 `Getter` 函数，一旦用户使用 `Symbol` 来访问对象中的属性，那么进入 `Getter` 以及 `Query` 函数时的“将 property 转为字符串”一步就会出错。事实上当我们对导出来的对象进行 `console.log` 时，Node.js 就会去访问它的一些 `Symbol`。所以正确的做法是，在 `Getter` 的第一行加上一句 `property` 是否是字符串的判断，如果不是则直接返回，好让 Node.js 继续访问这个对象自身的该属性。新增的代码如下：

```
NAN_PROPERTY_GETTER(Getter)
{
    if(!property->IsString()) return;

    ...
}

NAN_PROPERTY_QUERY(Query)
{
    if(!property->IsString()) return;

    ...
}
```

5.3.4 小结

本节主要介绍了 NAN 中设置函数的一些方法，包括：

- 普通函数声明与设置；
- 原型链函数的设置；
- 访问器函数的声明与设置；
- 拦截器函数的声明与设置。

除此之外，本节还通过样例介绍了如何将原生的 Node.js 扩展源码改成使用 NAN 来写的源码，以及提点了几个需要注意的地方。例如在设置映射型拦截器的时候不再有 `PropertyHandlerFlags` 等。

大家具备了函数设置的这些基础知识后，对于后续章节的理解来说是非常有帮助的。

除了这些常用内容之外的其他函数设置相关信息，以及最新的 NAN 信息，读者可以通过访问 NAN 的官方源码仓库获得。

5.3.5 参考资料

[1] nan/methods.md - JavaScript-accessible methods: <https://github.com/nodejs/nan/blob/master/doc/methods.md>.

5.4 常用帮助类与函数

在介绍了 NAN 写扩展的基石之后，本节将为大家呈现 NAN 中常用的一些帮助类与帮助函数。通过它们来代替 Node.js 和 Chrome V8 的原生内容，可以使开发者开发出来的扩展更具有兼容性。

5.4.1 句柄相关

1. 句柄作用域

细心的读者可能在随书代码“22. nan set method”的 `interceptor/addon.cc` 中就发现了，原本对于可逃句柄作用域的声明全都改用了 `Nan::EscapableScope` 来代替。

由于新旧版本 Node.js 的句柄作用域声明方式不同，因此在 NAN 中对它们也做了一个封装。最大的不同就是，NAN 中的句柄作用域不用再传 Isolate 实例了。

在 Chrome V8 中，除了通过 `args.GetIsolate()` 来获取 Isolate 实例，还能直接通过 `Isolate::GetCurrent()` 来获取。所以在 NAN 封装后的句柄作用域的构造函数中，是不需要传 Isolate 实例的，因为在其声明真正的句柄作用域时，是直接使用的 `Isolate::GetCurrent()`。

对于其封装的代码可以参考这里：

```
class HandleScope {
    v8::HandleScope scope;

public:
    #if NODE_MODULE_VERSION > NODE_0_10_MODULE_VERSION
        inline HandleScope() : scope(v8::Isolate::GetCurrent()) {}
        ...
    #else
        inline HandleScope() : scope() {}
        ...
    #endif
}
```

所以与原生 Node.js C++ 扩展开发相比，若使用 NAN 来声明句柄作用域，则直接如这段代码这样即可。

```
void SomeFunction()
{
    Nan::HandleScope scope;

    ...
}

Local<Value> SomeEscapableFunction()
{
    Nan::EscapableHandleScope scope;

    ...
}
```

2. 持久句柄

由于 Chrome V8 的接口变更，因此 NAN 为其持久句柄也封装了一层。于是，平时在读者使用 NAN 进行开发的时候，通常会使用 `Nan::Persistent` 而不再是 `v8::Persistent`，但是它持有的 API 与读者当前熟知的 `v8::Persistent` 所持有的 API 相差无几。

那么，与此类似，一些弱持久句柄操作的类型也都是使用 NAN 封装的，如 `Nan::WeakCallbackInfo` 和 `Nan::WeakCallbackType`。

例如：

```
static Nan::Persistent<v8::String> persistent;

NAN_METHOD(SomeFunc)
{
    // 将一个 Local<String> 升格为持久句柄
    persistent.Reset(info[0].As<v8::String>());

    // 构造函数将一个 Local<String> 升格为持久句柄
    Persistent<v8::String> test(info[0].As<v8::String>());

    // 获得一个持久句柄的本地句柄
    Local<v8::String> str = Nan::New(test);

    ...
}
```

5.4.2 创建数据对象

使用 NAN 创建数据对象，主要是通过 `Nan::New()` 这个函数进行的。它有一些元数据类型重载，根据传入不同的类型，直接返回你想要的结果；也有一些模板的重载，指定数据类型进行返回，如 `Nan::New<v8::String>("2333")`。

1. 创建元数据

在 NAN 中，3 种元类型数据（布尔类型、数值类型、字符串类型）是可以通过 `Nan::New` 的非模板函数的方式新建数据的。它们所需要的参数类型分别可以是布尔型、数值（`double`、`int` 等）以及字符串指针等。

不过值得注意的一点是，布尔类型和数值类型的数据通过 `Nan::New` 生成之后，直接就是一个本地句柄，而字符串返回的是待实本地句柄。

下面举个例子：

```
Local<Number> number = Nan::New(2333);
Local<Boolean> boolean = Nan::New(true);
Local<String> str = Nan::New("string").ToLocalChecked();
```

除了这 3 种数据类型之外，正则表达式类型也能通过这个便捷函数生成。因为 `Nan::New` 系列函数可接收多参数，所以其实它的函数声明类似于下面这样：

```
Nan::New<T>(A0 arg0);
Nan::New<T>(A0 arg0, A1 arg1);
Nan::New<T>(A0 arg0, A1 arg1, A2 arg2);
Nan::New<T>(A0 arg0, A1 arg1, A2 arg2, A3 arg3);
```

而在正则表达式生成的函数中，第一个参数类型是 `Local<String>`，第二个参数则是 `v8::RegExp::Flags`。

2. 创建本地句柄数据和待实本地句柄数据

除了在前文中笔者提到的一些便捷函数，在 NAN 中，读者要生成一些 V8 特有的数据句柄也还是通过 `Nan::New` 函数，只不过它是模板函数。

比如我们要生成一个数值型，严格来说应该使用 `Nan::New<v8::Number>(2333)`；要生成一个对象，则是 `Nan::New<v8::Object>()`。

总之，大家能想象得到的一些 Chrome V8 数据类型，通常都能通过 `Nan::New<T>()` 这个模板函数来生成，只不过是内部的参数类型、参数数量不同罢了。

在新建数据句柄的时候，有一些类型的数据返回的直接就是本地句柄，而还有一些类型的函数则是返回待实本地句柄。

比如在前文中提到的字符串，`Nan::New` 返回的就是一个待实本地句柄。读者如果要真正使用，则需要通过 `ToLocalChecked` 进行本地化。还有就是通过 `Nan::New<T>` 生成这些类型的时候返回的是待实本地句柄：

- `v8::String`
- `v8::Date`
- `v8::RegExp`
- `v8::Script`
- `v8::UnboundScript`

3. 创建特殊数据

所谓的特殊数据，就类似于 `undefined` 等类型。除了使用 `Nan::New` 进行 Chrome V8 的数据创建之外，NAN 还提供了下面几种方法供大家使用。

- `Nan::Undefined()`: 创建一个 `undefined`。
- `Nan::Null()`: 创建一个 `null`。
- `Nan::True()`: 创建一个布尔 `true`。
- `Nan::False()`: 创建一个布尔 `false`。
- `Nan::EmptyString()`: 创建一个空字符串。

5.4.3 与数据对象“玩耍”

NAN 除了负责创建数据对象之外，还负责一些常用的数据对象帮助函数，可以让读者跟 C++ 原生模块愉快地“玩耍”。

1. 类型转换

众所周知，Chrome V8 中的基础数据类型是 `Value`，它代表一个数据的抽象类型。但是我们在日常开发中，通常用它只是作为一个载体来传参数，而真正用到的是通过它转换而得的一些真实数据类型。

在 Chrome V8 的原生函数中，我们通常通过调用一些本地句柄的 `As<T>()` 函数等进行数据类型的转换。而在 NAN 的帮助下，我们是通过 `Nan::To()` 来进行数据转换的。

比如，要将一个 Chrome V8 的 `Value` 类型转换成一个布尔类型的本地句柄，就可以调用 `Nan::To<v8::Boolean>(你的句柄)` 来完成。如果想要更加地一步到位，将它转换成 C++ 下的布尔类型，也可以直接调用 `Nan::To<bool>(你的句柄)` 来完成。

如果你想要转换的结果是一个 Chrome V8 的数据类型，则函数的返回结果是一个用 NAN 封装的待实本地句柄；而如果你想要转换的结果是一个 C++ 自身的数据类型，则返回的是一个用 NAN 封装的 `Maybe1` 数据类型，其持有的 API 与 Chrome V8 的 `Maybe` 类型基本一致。

这是两种不同类型的类型转换器的函数声明，供大家参考。

```
// Chrome V8 类型转换器
Nan::MaybeLocal<v8::Boolean> Nan::To<v8::Boolean>(v8::Local<v8::Value> val);
Nan::MaybeLocal<v8::Int32> Nan::To<v8::Int32>(v8::Local<v8::Value> val);
Nan::MaybeLocal<v8::Integer> Nan::To<v8::Integer>(v8::Local<v8::Value> val);
Nan::MaybeLocal<v8::Object> Nan::To<v8::Object>(v8::Local<v8::Value> val);
Nan::MaybeLocal<v8::Number> Nan::To<v8::Number>(v8::Local<v8::Value> val);
Nan::MaybeLocal<v8::String> Nan::To<v8::String>(v8::Local<v8::Value> val);
Nan::MaybeLocal<v8::Uint32> Nan::To<v8::Uint32>(v8::Local<v8::Value> val);
```

¹ `Maybe` 是一种用于确认数据结果的模板类，在本书的 3.7.5 节中初次提到。

```
// C++ 原生类型转换器
Nan::Maybe<bool> Nan::To<bool>(v8::Local<v8::Value> val);
Nan::Maybe<double> Nan::To<double>(v8::Local<v8::Value> val);
Nan::Maybe<int32_t> Nan::To<int32_t>(v8::Local<v8::Value> val);
Nan::Maybe<int64_t> Nan::To<int64_t>(v8::Local<v8::Value> val);
Nan::Maybe<uint32_t> Nan::To<uint32_t>(v8::Local<v8::Value> val);
```

下面举个例子：

```
v8::Local<v8::Value> val;
Nan::MaybeLocal<v8::String> str = Nan::To<v8::String>(val);
Nan::Maybe<double> d = Nan::To<double>(val);
```

2. 和字符串“玩”

从函数参数中拿到一个 Chrome V8 的字符串很容易，无非是把某个参数通过 `Nan::To` 将其转换过来。不过，如果要将其转换为我们能直接用的 C++ 下的字符串的话，还是要做一些工作的。

这就好像直接基于 Chrome V8 进行开发一样，我们用了一个 `v8::Utf8Value` 的数据类型。而 NAN 同样给我们封装了一个 `Utf8String`。它的常用用法跟 `v8::String::Utf8Value` 很像，类的基本声明如下：

```
class Nan::Utf8String {
public:
    Nan::Utf8String(v8::Local<v8::Value> from);

    int length() const;

    char* operator*();
    const char* operator*() const;
};
```

比如，我们做一个字符串翻转的函数（参考随书代码的“23. nan reverse string”）。

```
NAN_METHOD(Reverse)
{
    Nan::MaybeLocal<String> handle = Nan::To<String>(info[0]);
    Local<String> local_handle = handle.ToLocalChecked();

    Nan::Utf8String val(local_handle);
```

```

// 字符串翻转
std::string str(*val);
std::reverse(str.begin(), str.end());

info.GetReturnValue().Set(Nan::New(str.c_str()).ToLocalChecked());
}

NAN_MODULE_INIT(Init)
{
    Nan::Export(target, "reverse", Reverse);
}

```

这段代码之初演示了 `Nan::New()` 和 `Nan::To()`，并给大家看了 `Nan::MaybeLocal` 待实句柄也一样有 `ToLocalChecked()` 等各种函数。接下来就是新建一个 `val` 的 `Nan::Utf8String` 对象，通过 `*` 操作符将其转换为 C 语言下的字符串，然后通过 `std` 的函数进行翻转。最后将得到的结果返回。

下面大家跑一跑试试看吧。

```

$ node-gyp rebuild
...
$ node
> const addon = require("./build/Release/reverse")
undefined
> addon.reverse("EggFlowerSoup");
'puoSrewolFggE'
> addon.reverse("蛋花汤");
'  汉苛  '

```

看样子一切运行正常。等等，最后一行的乱码是怎么回事？因为到了 `std::string` 中，一切都以 `char` 为准了啊，这里并没有 UTF8 的概念了，8 位的字符逐一翻转，自然拼出的是另一番景象了。

3. 和对象“耍”

在 Chrome V8 中，`v8::Object` 类型有一大堆的方法，比如 `Set`、`Get`、`Delete` 等。NAN 里面也为对象封装了这一系列的方法，不过都是接在 `Nan` 这个命名空间下面的，而并非作为 `Object` 的成员函数存在。

这些被封装的函数除了多了第一个参数是需要被操作的对象外，其余参数基本与 `v8::Object` 的相应成员函数一致。如 `Nan::Set(obj, key, value)` 对应的是 `obj->Set`

(key, value), 而返回值从 `v8::MaybeLocal` 或者 `v8::Maybe` 变成了 `Nan::MaybeLocal` 或者 `Nan::Maybe`。这些函数如下:

- `Nan::Maybe<bool> Nan::Set(...)`
- `Nan::Maybe<bool> Nan::ForceSet(...)`
- `Nan::MaybeLocal<Value> Nan::Get(...)`
- `Nan::Maybe<PropertyAttribute> Nan::GetPropertyAttributes(...)`
- `Nan::Maybe<bool> Nan::Has(...)`
- `Nan::Maybe<bool> Nan::Delete(...)`
- `Nan::MaybeLocal<Array> Nan::GetPropertyNames(...)`
- `Nan::MaybeLocal<Array> Nan::GetOwnPropertyNames(...)`
- `Nan::Maybe<bool> Nan::SetPrototype(...)`
- `Nan::MaybeLocal<String> Nan::ObjectProtoToString(...)`
- `Nan::Maybe<bool> Nan::HasOwnProperty(...)`
- `Nan::Maybe<bool> Nan::HasRealNamedProperty(...)`
- `Nan::Maybe<bool> Nan::HasRealIndexedProperty(...)`
- `Nan::Maybe<bool> Nan::HasRealNamedCallbackProperty(...)`
- `Nan::MaybeLocal<Value> Nan::GetRealNamedPropertyInPrototypeChain(...)`
- `Nan::MaybeLocal<Value> Nan::GetRealNamedProperty(...)`
- `Nan::MaybeLocal<Value> Nan::CallAsFunction(...)`
- `Nan::MaybeLocal<Value> Nan::CallAsConstructor(...)`
- `Nan::Maybe<bool> Nan::HasPrivate(...)`
- `Nan::MaybeLocal<Value> Nan::GetPrivate(...)`
- `Nan::Maybe<bool> Nan::SetPrivate(...)`
- `Nan::Maybe<bool> Nan::DeletePrivate(...)`

接下来还是进入实战环节吧, 打开随书代码“24. nan object demo”中的 `object.cc`, 看一下源码。

```
NAN_METHOD(Calc)
{
    if(!info.Length()) return;

    auto input = info[0]->ToObject();
```

```

auto output = Nan::New<Object>();

auto x_prop = Nan::New("x").ToLocalChecked();
auto y_prop = Nan::New("y").ToLocalChecked();
auto sum_prop = Nan::New("sum").ToLocalChecked();
auto product_prop = Nan::New("product").ToLocalChecked();

double x = Nan::Get(input, x_prop).ToLocalChecked()->NumberValue();
double y = Nan::Get(input, y_prop).ToLocalChecked()->NumberValue();

Nan::Set(output, sum_prop, Nan::New(x + y));
Nan::Set(output, product_prop, Nan::New(x * y));

info.GetReturnValue().Set(output);
}

```

从参数对象中读取 x 和 y 的值，将其相加以及相乘，将结果返回到一个新对象的 `sum` 和 `product` 字段中。该例中用了我们比较常用的 `Nan::Set` 和 `Nan::Get`，并复习了一下 `Nan::New`。

下面还是进入我们熟悉的跑代码环节吧。

```

$ node-gyp rebuild
...
$ node
> const addon = require("./build/Release/object");
undefined
> addon.calc({ x: 211, y: 22 });
{ sum: 233, product: 4642 }

```

没什么悬念，返回的就是我们想要的结果。

4. 和数组“共舞”

对于数组来说，用到的函数也无非就是那么几个，其中好几个还是与 `Object` 共用函数名的重载函数。除了与 `Object` 共用的一些重载函数之外，`NAN` 中的数组还封装了这样的一些方法：

- `Nan::MaybeLocal<Object> Nan::CloneElementAt(...)`
- `Nan::MaybeLocal<Uint32> Nan::ToArrayIndex(v8::Local<v8::Value> val)`

其中 `ToArrayIndex` 并不是直接针对数组用的，它是将一个 `Chrome V8` 的数据类型转为 `Chrome V8` 下数组能用的无符号整型的下标类型。

在前文翻转了字符串，笔者在这次的示例中翻转一个数组吧。打开“25. nan array demo”中的 array.cc 一起来研习一下。

```
NAN_METHOD(Reverse)
{
    Local<Array> input = Local<Array>::Cast(info[0]);
    Local<Array> output = Nan::New<Array>();

    for(unsigned int i = 0, j = input->Length() - 1; i < input->Length();
        i++, j--)
    {
        Nan::Set(output, j, Nan::Get(input, i).ToLocalChecked());
    }

    info.GetReturnValue().Set(output);
}
```

在一开始的时候通过 `Local<Array>::Cast()` 将参数转换为数组。

注意：目前 NAN 针对 Chrome V8 的数组并没有 `Nan::To<T>()` 的转换器重载，所以需要调用 Chrome V8 自身的一些转换函数。

在新声明一个数组之后，通过一个循环将数组进行翻转，并最后返回。跑跑看吧。

```
$ node-gyp rebuild
...
$ node
> const addon = require("./build/Release/array")
undefined
> addon.reverse([ "蛋花汤", "南瓜饼", "没有来的土豆球" ]);
[ '没有来的土豆球', '南瓜饼', '蛋花汤' ]
```

5. 逗一逗 JSON

在第3章中，笔者提到了 Chrome V8 可以通过 JSON 解析器对 JSON 数据进行 `stringify` 和 `parse`，而在 NAN 中同样对 JSON 有一个封装。

不过值得一提的是，在 Chrome V8 的原生代码中，JSON 里面的 `Stringify` 和 `Parse` 都是静态函数，可以直接被调用；而 `Nan::JSON` 中的这两个函数则是成员函数，需要实例化对象才能使用。

下面介绍稍微复杂一点的样例。函数传入一个 JSON 字符串，返回的也是一个 JSON 字符串，但是 JSON 内部的所有数值型的节点都被加一，比如说 `'{"a": [232, "蛋花汤"], "b": {"c": 232}}'` 就要返回 `{"a": [233, "蛋花汤"], "b": {"c": 233}}`。

我们一起打开“26. nan json”中的 `json.cc` 看看源码吧。先来看看 JavaScript 所调用的函数 `PlusOne`。

```
NAN_METHOD(PlusOne)
{
    Nan::JSON json;
    Nan::MaybeLocal<Value> parsed = json.Parse(Nan::To<String>(info[0]).
    ToLocalChecked());

    if(parsed.IsEmpty())
    {
        return info.GetReturnValue().Set(info[0]);
    }

    Local<Value> ret = parsed.ToLocalChecked();
    Plus(ret);

    info.GetReturnValue().Set(json.Stringify(Nan::To<Object>(ret).
    ToLocalChecked()).ToLocalChecked());
}
```

在该 `PlusOne` 函数中，先通过 `Nan::JSON` 将函数的第一个参数转换为一个 `Value` 类型赋值给 `ret`，然后将 `ret` 传入 `Plus` 函数中进行递归处理，最后将处理完毕的 `ret` 对象通过 `Nan::JSON` 重新字符串化并返回。

接下来开始分析这个递归的 `Plus` 函数。

```
void Plus(Local<Value>& obj)
{
    if(obj->IsNumber())
    {
        obj = Nan::New(Nan::To<double>(obj).FromJust() + 1);
        return;
    }
    if(!obj->IsObject()) return;

    if(obj->IsArray())
    {
        auto array = Local<Array>::Cast(obj);
        for(unsigned int i = 0; i < array->Length(); i++)
```

```

{
    Local<Value> val1;
    Local<Value> val2;
    val1 = val2 = Nan::Get(array, i).ToLocalChecked();

    Plus(val1);
    if(val1 != val2)
    {
        Nan::Set(array, i, val1);
    }
}
return;
}

Local<Object> object = Nan::To<Object>(obj).ToLocalChecked();
auto properties = Nan::GetOwnPropertyNames(object).ToLocalChecked();
for(unsigned int i = 0; i < properties->Length(); i++)
{
    Local<String> key = Nan::To<String>(Nan::Get(properties,
i).ToLocalChecked()).ToLocalChecked();
    Local<Value> val1;
    Local<Value> val2;
    val1 = val2 = Nan::Get(object, key).ToLocalChecked();

    Plus(val1);
    if(val1 != val2)
    {
        Nan::Set(object, key, val1);
    }
}
return;
}

```

如代码所示：

- ① 如果当前传入的值是一个数值类型的话，则将引用赋值成一个新的数值类型，其值是旧值加一。
- ② 若它既不是数值类型也不是对象的话，则直接返回。
- ③ 若它是数组（也是对象类型的一种）的话，则遍历数组元素，并对每个元素递归再调用 Plus 后重新赋值。
- ④ 若它是真对象的话，则遍历对象键名，并对每个对应的值再次调用 Plus 后重新赋值。

在这次的样例中分别使用了 `Nan::JSON` 的 `Parse` 和 `Stringify`，并且使大家复习了 NAN 对于对象、数组的一系列操作，以及数据的新建和转换。这可以说是“知识点满满”。闲话就不多说了，大家来看看运行结果吧。

```
$ node-gyp rebuild
...
$ node
> const addon = require("./build/Release/json");
undefined
> addon.plusOne("123");
'124'
> addon.plusOne("\123\");
'"123"'
> addon.plusOne('{ "a": [232, "蛋花汤"], "b": { "c": 232 } }');
'{"a": [233, "蛋花汤"], "b": { "c": 233 } }'
```

若 JSON 字符串所代表的内容直接是一个数字的话，返回的是它加 1 的值；若 JSON 字符串所代表的内容是一个字符串的话，原样返回；否则会递归遍历对象的每个元素，遇到数值类型的元素，则对其加 1。

6. 别忘了函数

当我们手头有一个函数 `Function` 的本地句柄的时候，我们可以通过 `Nan::Call()` 调用它。其函数声明如下：

```
Nan::MaybeLocal<Value> Nan::Call(Local<Function> fun, Local<Object> recv,
int argc, Local<Value> argv[]);
```

这看起来跟 `v8::Function` 的成员函数 `Call` 差不多。除此之外，与 `Nan::Call()` 类似的还有 `Nan::MakeCallback()`，这是一个语义上比 `Nan::Call()` 更精确的回调函数调用的函数。具体的例子大家可以往回翻看“21. nan array prototype map”和“22. nan set method”中的源码。

7. 还有一个 Buffer “小可爱”

`Buffer` 的确是一个“小可爱”，以至于它在本书的前面章节中几乎都没被提到——因为它本就不是 Chrome V8 的内容，而是在 Node.js 中被实现的。`Buffer` 的各种具体用法，可参照 Node.js 中相应版本的 `node_buffer.h` 和 `node_buffer.cc` 源码——实际上它就是对于计算机内存中一块内存块的封装。

在 NAN 中，对于 `Buffer` 的声明也做了一些简单的封装，让我们来看看它封装的一些函数吧。

- `Nan::MaybeLocal<Object> Nan::NewBuffer(uint32_t size)`

- `Nan::MaybeLocal<Object> Nan::NewBuffer(char* data, uint32_t size)`
- `Nan::MaybeLocal<v8::Object> Nan::NewBuffer(char *data, size_t length, Nan::FreeCallback callback, void *hint)`
- `Nan::MaybeLocal<v8::Object> Nan::CopyBuffer(const char *data, uint32_t size)`
- `typedef void (*FreeCallback)(char *data, void *hint)`

由于 Buffer 在 Node.js 中本质上是一个对象（即 Object），因此函数的返回值都是对象的待实本地句柄。

其中 `Nan::NewBuffer()` 函数是通过传入的内存块初始指针生成一个新的 Buffer，并且该内存块指针的生命周期应当移交给 Buffer 管理，而不应该手动释放。也就是说，当开发者没有在 `NewBuffer()` 时指定 `Nan::FreeCallback callback` 的时候，一旦垃圾回收机制触发回收这个 Buffer 对象，其一开始传进来作为内容本体的指针会被通过调用 `free()` 函数释放。如果这个时候开发者还一意孤行自行去释放指针的话，会引来很麻烦的后果。

而 `Nan::CopyBuffer()` 与 `Nan::NewBuffer()` 相似，不同的是它会在构造的时候隐式复制传入的内存块。这样一来，一开始传入的内存块的生命周期由开发者自行负责，需要自行在合适的时机释放。

最后一个 `Nan::FreeCallback` 类型是用于 `Nan::NewBuffer()` 中的一个重载的参数的。当一个用该重载的 Buffer 将要被释放的时候，会调用这个 callback 函数，开发者需要在里面做一些善后工作。

为了加深印象，笔者逐步解析一下“27. nan buffer”中 `buffer.cc` 里面的 3 个函数。

首先是 `PlusOne1` 和它的 `FreeCallback`。

```
void FreeCallback(char* buf, void* hint)
{
    printf("Free.\n");
    delete []buf;
    printf("Freed.\n");
}

NAN_METHOD(PlusOne1)
{
    Local<Object> old = Nan::To<Object>(info[0]).ToLocalChecked();

    unsigned int length = node::Buffer::Length(old);
    char* buf = node::Buffer::Data(old);
```

```

for(unsigned int i = 0; i < length; i++)
{
    buf[i] ^= 233;
}

info.GetReturnValue().Set(Nan::NewBuffer(
    buf,
    length,
    FreeCallback,
    0).ToLocalChecked());
}

```

在 JavaScript 层调用这个 `PlusOne1` 的时候，函数内部首先会得到一个 `Buffer` 对象并赋值给 `old`，接着得到它内部的真实内存块地址，然后对于每个元素都进行一次异或操作，最后通过 `Nan::NewBuffer()` 将得到的内存块再封装到一个新的 `Buffer` 中，并且 `FreeCallback` 为笔者事先写好的函数，即当这个 `Buffer` 要被垃圾回收时会调起的函数。

乍一看这段代码没什么问题，我们实际运行一遍看看结果吧。

```

$ node-gyp rebuild
...
$ node --expose_gc
> const addon = require("./build/Release/buffer");
undefined
> const temp = { foo: new Buffer(2) };
undefined
> temp.bar = addon.plusOne1(temp.foo);
<Buffer ed c9>
> temp
{ foo: <Buffer ed c9>, bar: <Buffer ed c9> }
> temp.foo[0] = 0;
0
> temp
{ foo: <Buffer 00 c9>, bar: <Buffer 00 c9> }

```

注意：为了方便演示，这次执行的时候加入了 `--expose_gc` 参数，可以在必要的时候通过 `gc()` 函数来强制垃圾回收。

我们在运行样例中先声明了一个 `temp.foo` 的 `Buffer`，然后通过 `addon.plusOne1` 弄出一个 `bar`。输出 `temp` 看的时候，发现 `temp.foo` 和 `temp.bar` 所对应的值是一样的。原因很简单，它们共用了同一个内存块，所以笔者在之后修改了 `temp.foo[0]` 的值的时候，`temp.bar` 也跟着改变了。

接着我们继续输入其他一些代码，执行后看看结果。

```
> delete temp.bar;
true
> gc();
Free.
node(29968,0x7fff992353c0) malloc: *** error for object 0x102055208: pointer
being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
[1] 29968 abort node --expose_gc
```

当我们执行上面这些内容的时候，问题就出来了。当我们执行 `delete temp.bar` 并且强制进行垃圾回收的时候，终端会先输出 `Free.` 这句内容，并且开始执行 `delete []buf;`，之后程序就崩溃在这里了。按照错误信息解释，这就是说，`buf` 并不是通过 `new` 或者 `alloc` 等方式声明的，所以无法通过 `delete` 或者 `free` 的方式进行释放。如果大家有兴趣的话，可以尝试把 `FreeCallback` 中的 `delete` 改成 `free`，以同样方式运行也会出现一样的崩溃。

注意：事实上 Node.js 中对于自身的 `Buffer` 内置内存块分配是通过 `realloc` 进行的，并且通过 `Chrome V8` 的 `ArrayBuffer` 封装，所以我们不要对它暴露出来的内存块指针做一些不好的操作。

因为 `buf` 本身是 `temp.foo` 内置的内存块，而 `temp.bar` 中的内存块实际上是与 `temp.foo` 共用的，所以哪怕真的可以通过 `delete` 或者 `free` 进行回收，那么在回收之后，`temp.foo` 元素内置的内存块指针就变成了一个野指针，下次删除 `temp.foo` 的时候同样会对该指针进行一次删除，对野指针进行一些操作一样也会导致崩溃。所以笔者不得不说，这段代码是错误的。

接下来我们再来看看 `PlusOne2` 这个函数的内容吧。

```
NAN_METHOD(PlusOne2)
{
    Local<Object> old = Nan::To<Object>(info[0]).ToLocalChecked();

    unsigned int length = node::Buffer::Length(old);
    char* buf = node::Buffer::Data(old);

    for(unsigned int i = 0; i < length; i++)
    {
        buf[i] ^= 233;
    }

    info.GetReturnValue().Set(Nan::CopyBuffer(buf, length).
```

```
ToLocalChecked());
}
```

其前半部分与 `PlusOne1` 是一样的，都是对原内存块中的每个元素进行异或操作。最后在生成新 `Buffer` 的时候是通过 `Nan::CopyBuffer` 来进行的。也就是说被塞入新 `Buffer` 中的内存块不是与老的共用，而是在 `NAN` 中复制出一个新的内存区域，而老的内存块的生命周期并没有被接管，还是由老的 `Buffer` 自行管理；而新的 `Buffer` 内存块由新的 `Buffer` 管理，我们无须做任何多余的事情。

不过如果我们传给 `CopyBuffer()` 的是一个自己通过 `new` 得到的内存区域，则应该自行在适当的时机（比如 `Nan::CopyBuffer()` 这行代码之后）通过 `delete` 释放它。

解释到这里，差不多也该执行代码看看了，还是先前的执行步骤。

```
$ node-gyp rebuild
...
$ node --expose_gc
> const addon = require("./build/Release/buffer");
undefined
> const temp = { foo: new Buffer(2) };
undefined
> temp.bar = addon.plusOne1(temp.foo);
<Buffer ed c9>
> temp
{ foo: <Buffer ed c9>, bar: <Buffer ed c9> }
> temp.foo[0] = 0;
0
> temp
{ foo: <Buffer 00 c9>, bar: <Buffer ed c9> }
> delete temp.bar;
true
> gc();
undefined
> delete temp.foo;
true
> gc();
undefined
```

这就是 `Nan::CopyBuffer()` 的正确使用姿势了。

最后，笔者再来讲讲 `PlusOne3`。

```
NAN_METHOD(PlusOne3)
{
```



```

Local<Object> old = Nan::To<Object>(info[0]).ToLocalChecked();

unsigned int length = node::Buffer::Length(old);
char* buf = new char[length];
memcpy(buf, node::Buffer::Data(old), length);
for(unsigned int i = 0; i < length; i++)
{
    buf[i] ^= 233;
}

info.GetReturnValue().Set(Nan::NewBuffer(
    buf,
    length,
    FreeCallback,
    0).ToLocalChecked());
}

```

在 `PlusOne3` 中，笔者自行复制了旧 `Buffer` 的内存块，并将复制后的内存块内容进行异或，接着还是通过 `Nan::NewBuffer()` 生成新的 `Buffer` 返回。新内存块的生命周期结束回调由 `FreeCallback` 接手，该函数内部会将这个内存块指针进行 `delete` 操作。

下面继续按照先前的步骤执行一遍吧。

```

$ node-gyp rebuild
...
$ node --expose_gc
> const addon = require("./build/Release/buffer");
undefined
> const temp = { foo: new Buffer(2) };
undefined
> temp.bar = addon.plusOne3(temp.foo);
<Buffer e9 c9>
> temp
{ foo: <Buffer 00 20>, bar: <Buffer e9 c9> }
> delete temp.bar;
true
> gc();
Free.
Freed.
undefined
> delete temp.foo;
true
> gc();
undefined

```

从结果中我们看到，在生成新 Buffer 后，新老 Buffer 中的内容已经不一样了，老的 Buffer 中的内容保持着它刚生成的样子，而新的里面则是经过异或操作后的样子。在我们执行 `delete temp.bar` 并且执行 `gc()` 的时候，FreeCallback 会顺利执行，成功释放我们自行复制出来的内存块。在删除 `temp.foo` 并且强制进行垃圾回收的时候，也是异常顺利。

至此，关于 Buffer “小可爱”的 3 个操作就介绍完了。

5.4.4 封装一个类

在 NAN 中封装一个类也是非常简单的，几乎可以照搬 4.3.1 节中封装一个类的写法；不同的是，我们原本继承自 `node::ObjectWrap` 的类应该改为继承自 `Nan::ObjectWrap`。

还需要注意的是，原本在类中的一些成员函数和静态函数的声明，最好也应该使用 `NAN_METHOD`、`NAN_MODULE_INIT` 等宏来代替。

例如在笔者随书代码“18. myobject”中的 `1/myobject.h` 对于 `MyObject` 的声明应该改为这样：

```
class MyObject : public Nan::ObjectWrap {
public:
    static NAN_MODULE_INIT(Init);

private:
    explicit MyObject(double value = 0);
    ~MyObject();

    static NAN_METHOD(New);
    static NAN_METHOD(PlusOne);
    double value_;
};
```

而其中的一些实现如对于函数、对象等的设置和获取也应当改为 NAN 提供的方法。

还有值得注意的一点，就是往对象里面塞内置字段的时候，也应该使用 NAN 提供的方法来代替。

- `void* Nan::GetInternalFieldPointer(Local<Object> object, int index)`
- `void Nan::SetInternalFieldPointer(Local<Object> object, int index, void* value)`

大家敞开“脑洞”，试着改改“18. myobject”里面的代码吧。

5.4.5 异常处理

除了笔者先前讲的那么多特性之外，NAN 还把它的“魔爪”伸向了异常处理。下面先来看看它有哪些好用的方法吧：

- `Nan::Error()`
- `Nan::RangeError()`
- `Nan::ReferenceError()`
- `Nan::SyntaxError()`
- `Nan::TypeError()`
- `Nan::ThrowError()`
- `Nan::ThrowRangeError()`
- `Nan::ThrowReferenceError()`
- `Nan::ThrowSyntaxError()`
- `Nan::ThrowTypeError()`
- `Nan::FatalException()`
- `Nan::ErrnoException()`
- `Nan::TryCatch`

从 `Nan::Error()` 到 `Nan::TypeError()` 这几个函数都是用于生成一个错误实例的。这几个函数都接收两种类型的参数：一种是 `const char*` 字符串，另一种是 `Local<String>`，两类参数都代表错误信息。

在上面所列的函数中，自 `Nan::ThrowError()` 开始到 `Nan::ThrowTypeError()` 的这几个函数都代表往 JavaScript 抛出一个错误，除了接收与 `Nan::Error()` 等函数相同的参数之外，它们还有一个重载，就是接收一个错误对象（`Local<Value>`）。

`Nan::FatalException()` 和 `Nan::ErrnoException()` 两个函数分别是 node 命名空间下两个同名函数的替身，有兴趣的读者可以自行查看 Node.js 源码，阅读这两个函数的作用。

5.4.6 小结

本节介绍了 NAN 作为 Node.js 原生模块开发的替身，都做了哪些替换。大到函数声明，小到对象成员，都有一些相应的函数。不过，由于篇幅原因和时效性的原因，本节并没有完全地介绍所有 NAN API，而只是让大家对其开发有一个印象。

如果大家需要真正投入开发的话，则应该实时打开 NAN 的官方文档进行阅读并开始你的开发工作。

5.4.7 参考资料

- [1] nan/scopes.md - Scopes: <https://github.com/nodejs/nan/blob/master/doc/scopes.md>.
- [2] nan/persistent.md - Persistent references: <https://github.com/nodejs/nan/blob/master/doc/persistent.md>.
- [3] nan/new.md - New: <https://github.com/nodejs/nan/blob/master/doc/new.md>.
- [4] Scott Frees. C++ and Node.js Integration[EB/OL]. <https://scottfrees.com/ebooks/nodecpp/>.
- [5] nan/new.md - Converters: <https://github.com/nodejs/nan/blob/master/doc/converters.md>.
- [6] nan/maybe_types - Maybe Types: https://github.com/nodejs/nan/blob/master/doc/maybe_types.md.
- [7] nan/buffers - Buffers: <https://github.com/nodejs/nan/blob/master/doc/buffers.md>.

5.5 NAN 中的异步机制

NAN 除了对于 Node.js 和 V8 进行了一些基础的封装之外，还封装了一些简单易用的异步帮助类。这些帮助类是基于 libuv 的。虽然在第 6 章会对使用 libuv 进行异步扩展开发进行介绍，但是对于简单的需求来说，使用 NAN 的异步帮助类就足够了，最重要的是它简单、方便。

在 NAN 中，主要使用 `AsyncWorker`、`AsyncProgressWorker` 两个类做一些异步的事情。

5.5.1 `Nan::AsyncQueueWorker`

在介绍 `AsyncWorker` 和 `AsyncProgressWorker` 之前，笔者在此先介绍一下它们的相关函数 `AsyncQueueWorker`。

```
void AsyncQueueWorker(AsyncWorker *);
```

这个函数是将 `AsyncWorker` 的对象加入自己内部，并且异步去执行相关任务用的。举一个最简单的例子，比如笔者有一个 `MyWorker` 的类继承自 `AsyncWorker`，并且笔者现在想执行它的异步任务，就直接这么调用：

```
// 假设 `callback` 是回调函数
AsyncQueueWorker(new MyWorker(callback));
```

5.5.2 Nan::Callback

除了 `Nan::AsyncQueueWorker` 函数之外，读者在 NAN 编写异步代码的时候，还需要了解的内容是一个 NAN 封装的类——`Callback`。

`Nan::Callback` 的出现是为了让开发者在将一个 `v8::Function` 的函数句柄当作回调函数使用时更为方便。这个类有以下特性：

- 将 `v8::Function` 句柄封装在了自身内部。
- 保护它不受 Chrome V8 垃圾回收的迫害。
- 有能力存储一些必要的的数据。
- 跨异步执行的能力。

关于这个类的具体声明，有兴趣的读者可以阅读该类的文档。在本节中，着重给大家指出两个重要的成员函数。

- ① 构造函数：`Nan::Callback` 的构造函数需要传入的参数是一个 `v8::Function` 的本地句柄。
- ② `Call(...)`：执行封装的函数，它有两个重载。

(a) `Local<Value> Call(Local<Object> target, int argc, Local<Value> argv[])`

(b) `Local<Value> Call(int argc, Local<Value> argv[])`

5.5.3 Nan::AsyncWorker

`AsyncWorker` 是 NAN 中用于异步操作的基类之一。我们来看看这个类的基础声明（去除一些杂项的成员）。

```
class AsyncWorker {
public:
    explicit AsyncWorker(Callback *callback_);
    virtual ~AsyncWorker();

    virtual void WorkComplete();

    void SaveToPersistent(const char *key, const v8::Local<v8::Value> &value);
```

```

void SaveToPersistent(const v8::Local<v8::String> &key,
                     const v8::Local<v8::Value> &value);
void SaveToPersistent(uint32_t index,
                     const v8::Local<v8::Value> &value);

v8::Local<v8::Value> GetFromPersistent(const char *key) const;
v8::Local<v8::Value> GetFromPersistent(const v8::Local<v8::String> &key)
const;
v8::Local<v8::Value> GetFromPersistent(uint32_t index) const;

virtual void Execute() = 0;
virtual void Destroy();

protected:
Persistent<v8::Object> persistentHandle;
virtual void HandleOKCallback();
virtual void HandleErrorCallback();
void SetErrorMessage(const char *msg);
const char* ErrorMessage();
};

```

该类的构造函数中传入的 Callback 就是异步执行完毕之后所需要执行的回调函数。

在我们实现自己的异步类时，最需要实现的虚函数是 Execute() 和 HandleOKCallback()。而剩下的 HandleErrorCallback() 和 Destroy() 等函数可以视情况实现。

这里还有几个需要稍稍提一下的函数是 SaveToPersistent() 系列函数和 GetFromPersistent()。NAN 的 AsyncWorker 中可以通过 SaveToPersistent() 将一些本地句柄转换成持久句柄存储在自己内部，用于防止这些数据由于异步和作用域而被 Chrome V8 的垃圾回收机制回收；而 GetFromPersistent() 则是用于获取被转换的持久句柄并转回本地句柄。

1. Execute()

我们在继承 AsyncWorker 时，最需要实现的就是 Execute() 函数了。它是异步执行的主体。

在 NAN 需要执行该异步任务的时候，会通过 libuv 的 uv_queue_work 函数唤起线程池的调度，其中在线程中异步执行期间，执行的就是 Execute() 函数了。

下面举个例子，如果读者想在 Node.js 的工作线程池中执行 2.1.1 节中提到的 50 000 000 次模拟的话，就可以写一个 NBodyWorker 并实现它的 Execute() 函数。

```
void NBodyWorker::Execute()
{
    NBodySystem bodies;
    for(int i = 0; i < n; i++) bodies.advance(0.01);

    // 假设 `energy` 是 `NBodyWorker` 的一个成员变量
    energy = bodies.energy();
}
```

这样，当读者执行 `AsyncQueueWorker(new NBodyWorker(callback))` 时，这个 `Execute` 中的模拟操作就会被 `libuv` 拉到线程池中执行了。

注意：在 `libuv` 以及使用它的 `Node.js` 中，线程池中的线程是有上限的，默认为 4¹。所以当我们的计算任务过重时，就有可能阻塞住所有的异步线程，这个时候其他的异步任务也一样会被阻塞等待，从而降低程序执行效率。因此，哪怕是重型任务被改成异步操作，也一样无法改变 `Node.js` 实际上对于应付计算密集型任务时的短板。

另外，这个函数是在 `Worker` 线程中执行的，所以不能安全访问 `Chrome V8` 及其数据对象。因此在这个函数内我们最好不要获取和操作各种 `Chrome V8` 的对象，我们在这个函数中要做的仅仅只是对 `C++` 底层数据的输入和输出以及操作。²

2. SetErrorMessage()

这个函数是一个 `protected` 成员，主要在 `Execute()` 的实现中被调用。当读者实现 `Execute()` 时候，如果觉得哪一步发生了异常，就可以通过 `SetErrorMessage()` 函数来设置错误。这样一来，在 `NAN` 最后执行回调的时候，就会通过判断是否有错误被设置而选择执行 `HandleOKCallback()` 和 `HandleErrorCallback()` 了。

3. HandleOKCallback() 和 HandleErrorCallback()

这两个函数会在 `Execute()` 函数执行完之后，通过 `libuv` 的调度，在 `Node.js` 主事件循环中被调用。它们并非纯虚函数，`NAN` 会通过判断 `Execute()` 执行完毕之后是否有被设置错误对象来选择执行，原函数内部的实现是这样的。

- `HandleOKCallback()`：直接调用 `callback`，不传任何参数。
- `HandleErrorCallback()`：直接调用 `callback`，并将通过 `SetErrorMessage()` 设置的错误信息给实例化成 `Chrome V8` 的错误对象，然后作为 `callback` 的第一个参数传入。

¹ `libuv` 中线程池的线程默认数量为 4，但是可以通过设置 `UV_THREADPOOL_SIZE` 这个环境变量来扩充线程数，不过不超过 128。在 `libuv` 的文档中及 `Node.js` 文档中的 https://nodejs.org/docs/v6.9.4/api/dns.html#dns_dns_lookup 都曾提及。

² 该提示在 https://github.com/nodejs/nan/blob/v2.6.2/examples/async_pi_estimate/async.cc#L31-L34 中被提及。

如果开发者觉得这两个函数的逻辑太简单而无法满足自身的需求, 就可以选择自行实现。例如笔者先前讲到的 `NBodyWorker`, 在通常的认知里, 理应将计算好的 `energy` 传入回调函数, 也就是说应该这样实现 `HandleOKCallback()`:

```
void HandleOKCallback()
{
    // 在这里没有外层的句柄作用域包装了,
    // 所以在该函数内记得自行声明作用域
    Nan::HandleScope scope;

    Local<Value> argv[] = { Nan::Null, Nan::New<Number>(energy) };
    callback->Call(2, argv);
}
```

正如代码所示, 这样一来, 在 `Execute()` 中计算出来的 `energy` 就被传入了回调函数中。

4. NBody 示例

`NBody` 问题是使用简易的辛积分器来模拟类木行星的轨道的一个问题, 在 2.1.1 节中有具体说明。在本节中笔者将对 `NBody` 的 C++ 代码做一层 NAN 的异步封装。

事已至此, 我们打开 “28. nan nbodies” 的 `addon.cc` 来看看吧。

下面先来看看 `Calc` 函数。

```
NAN_METHOD(Calc)
{
    Nan::Callback* callback;
    unsigned int times = 50000000;
    if(info.Length() > 1)
    {
        double _times = Nan::To<double>(info[0]).FromJust();
        if(_times < 0)
        {
            Nan::ThrowRangeError("Wrong times.");
            return;
        }

        times = _times;
        callback = new Nan::Callback(info[1].As<Function>());
    }
    else
    if(info.Length() > 0)
    {
        callback = new Nan::Callback(info[0].As<Function>());
    }
```



```

    }
    else
    {
        Nan::ThrowTypeError("Callback needed.");
    }

    Nan::AsyncQueueWorker(new NBodyWorker(times, callback));
}

```

不考虑健壮性，该段代码在有两个以上参数时，使用第一个参数作为模拟次数，使用第二个参数作为回调函数。如果只有一个参数的话，则参数作为回调函数，模拟次数为默认的 50 000 000，否则抛错。

在得到了模拟次数之后，笔者从堆中创建一个 `NBodyWorker` 并将其通过 `AsyncQueueWorker` 加入异步队列中让它自执行。

接下来看一下这个 `NBodyWorker` 类。

```

class NBodyWorker : public Nan::AsyncWorker {
public:
    NBodyWorker(unsigned int times, Nan::Callback* callback) :
        times(times),
        energy(0),
        Nan::AsyncWorker(callback)
    {
    }

    ~NBodyWorker() {}

    void Execute()
    {
        NBodySystem bodies;
        for(unsigned int i = 0; i < times; i++) bodies.advance(0.01);
        energy = bodies.energy();
    }

    void HandleOKCallback()
    {
        Nan::HandleScope scope;

        Local<Value> argv[2] = {
            Nan::Undefined(),
            Nan::New(energy)
        };
        callback->Call(2, argv);
    }
}

```

```

    }

private:
    unsigned int times;
    double energy;
};

```

在构造函数中，将 `callback` 传给它的父类进行构造。然后 `Execute()` 函数的实现就如笔者先前讲的一样，通过一个 `for` 循环来模拟，最后得到 `energy`。

最后在 `HandleOKCallback` 中，我们将得到的 `energy` 封装成一个 `v8::Number`，之后执行 `callback` 将结果传递出去。

是不是觉得在已有 C++ 的源码情况下，只做一个封装远比自己“扒”代码重新用 JavaScript 写一遍要简单得多？（当然，这里笔者假设并没有 JavaScript 版本的代码。）

好了，接下来看看执行结果吧。

```

$ node-gyp rebuild
...
$ node
> const addon = require("./build/Release/nbody");
undefined
> addon.calc(0, function() { console.log(arguments); });
undefined
# 过一段时间
{ '0': undefined, '1': -0.16907516382852447 }
> addon.calc(1000, function() { console.log(arguments); });
undefined
# 过一段时间
{ '0': undefined, '1': -0.16908760523460542 }
> console.time(50000000); addon.calc(function() { console.timeEnd(50000000);
console.log(arguments); });
undefined
# 过一段时间
50000000: 5000.160ms
{ '0': undefined, '1': -0.1690599068117857 }

```

从执行结果中，我们能看到当 `times` 为 0 和 1000 时，模拟结果分别约是 -0.169075164 和 -0.169087605，结果跟正确结果¹一致。然后当 `times` 为 50 000 000 时，在笔者的计算机上执行结果约为 5 秒，并且还是异步不阻塞主事件循环的。

1 <https://benchmarksgame.alioth.debian.org/download/nbody-output.txt>

同样的算法，当使用 JavaScript 版的代码时，在笔者的计算机上模拟 50 000 000 次的时间是 7 秒左右。

5.5.4 Nan::AsyncProgressWorker

AsyncProgressWorker 实际上是 AsyncProgressWorkerBase<char>，它继承自 AsyncWorker，其中模板定义的 <char> 代表传给 HandleProgressCallback 的参数类型。如果你不满足于使用 char* 的类型来给该函数传递参数的话，可以让自己写的 Worker 继承自 AsyncProgressWorkerBase<T>。

1. HandleProgressCallback() 和 Execute()

AsyncProgressWorkerBase<T> 比 AsyncWorker 需要多实现一个 HandleProgressCallback(const T* data, size_t size) 函数，代表每次到处理点的时候 NAN 调用的函数。

除此之外，Execute 函数的声明也变了，现在是 void Execute(const ExecutionProgress& progress)。其中 ExecutionProgress& 这个类型的参数用于通过成员函数 Send(T* data, size_t size) 发起一次 HandleProgressCallback 函数的调用，即俗话说的“到点了要调用一次”。

2. 简易 HTTP 异步请求示例

与 AsyncWorker 相比，AsyncProgressWorker 并不常用，它主要用于那种中间可能会多次触发某种回调函数，并且在整个任务执行结束之后执行一次总的回调函数的场景。

如下载内容报进度，它的一个使用原型就可能是这样的：

```
let content = "";
addon.download("一个网址", function(data) {
    content += data.toString();
}, function(err) {
    if(err) console.error(err);
    else console.log(content);
});
```

或者还有一种场景，就是 Async¹ 这个库里面的 async.mapLimit 函数，只不过如果要实现跟 async.mapLimit 差不多的函数，我们还必须要借助 libuv 的力量，而无法直接通过 NAN 提供的异步帮助类来完成。

¹ JavaScript 解决异步函数“回调地狱”的库：<https://github.com/caolan/async>，笔者个人对它的喜好程度远大于 Promise 和 yield、await 等。

我们还是来模拟一个下载的函数吧，也就是上面的 `foo.download`。打开“29. nan dummy download”的 `download.cc` 吧。

下面先来看一下 `HttpDumpSocket` 类。

```
class HttpDumpSocket : public minihttp::HttpSocket {
public:
    HttpDumpSocket(const AsyncProgressWorker::ExecutionProgress& progress) :
        progress(&progress)
    {
    }
    virtual ~HttpDumpSocket() {}

protected:
    void _OnRecv(void* buf, unsigned int size)
    {
        if(!size) return;
        progress->Send((const char*)buf, size);
    }

private:
    const AsyncProgressWorker::ExecutionProgress* progress;
};
```

这个类参考自 `minihttp` 的 `example2.cpp`¹，大致意思就是说，当某个 HTTP 页面在接收数据的过程中，每获得一个 Buffer 就会调用一次 `_OnRecv()` 函数。而笔者做的就是实现这个 `_OnRecv()` 时在里面执行了一下 `progress->Send()` 来调起 `DownloadWorker` 的 `HandleProgressCallback()` 函数。

其次，就是笔者在本节讲的重点，对 `AsyncProgressWorker` 进行继承了。

```
class DownloadWorker : public AsyncProgressWorker {
public:
    DownloadWorker(string uri, Nan::Callback* callback, Nan::Callback*
processor) :
        AsyncProgressWorker(callback),
        processor(processor),
        uri(uri)
    {
    }

    ~DownloadWorker()
```

¹ <https://github.com/fgenesis/minihttp/blob/master/example2.cpp>

```

    {
        delete processor;
    }

void Execute(const AsyncProgressWorker::ExecutionProgress& progress)
{
    HttpDumpSocket ht(progress);
    ht.SetBufsizeIn(64 * 1024);
    ht.SetFollowRedirect(true);
    ht.SetUserAgent("minihttp");
    ht.SetNonBlocking(false);
    ht.SetAlwaysHandle(false);
    ht.Download(uri);
    while(ht.isOpen() || ht.HasPendingTask()) ht.update();
}

void HandleProgressCallback(const char* data, size_t count)
{
    Nan::HandleScope scope;

    Local<Value> argv[] = {
        Nan::CopyBuffer(data, count).ToLocalChecked()
    };
    processor->Call(1, argv);
}

private:
    Nan::Callback* processor;
    string uri;
};

```

在 `Execute()` 函数的实现中，笔者声明了一个刚才实现的 `HttpDumpSocket`，并且使用它进行 `Download(uri)` 操作。这样，Node.js 就会在这个工作线程中进行与目标网址的交互了。在每次有 `Buffer` 进来的时候，会执行一下刚才说的 `_OnRecv()` 函数。也就是最终会执行 `HandleProgressCallback()` 函数。在这个函数中，我们将收到的数据通过 `Nan::CopyBuffer()` 包装成一个 Node.js 中的 `Buffer`，并调用事先传入的 `processor`。

`Execute()` 会阻塞在它末尾的 `while` 循环中，直到请求结束。一旦 `Execute()` 结束，我们知道它就去执行 `HandleOKCallback()` 了。未经子类实现的 `HandleOKCallback()` 函数会直接调用 `callback`，并不传任何参数。

最后，我们再来看看实现的这个异步 `download` 函数。

```
NAN_METHOD(Download)
{
    Local<String> uri = Nan::To<String>(info[0]).ToLocalChecked();
    Nan::Utf8String curi(uri);

    Nan::Callback* processor = new Nan::Callback(info[1].As<Function>());
    Nan::Callback* callback = new Nan::Callback(info[2].As<Function>());

    Nan::AsyncQueueWorker(new DownloadWorker(*curi, callback, processor));
}
```

它拿到了传入的 3 个参数之后，一股脑儿地声明了一个 `DownloadWorker` 并通过 `AsyncQueueWorker()` 加入了异步队列中。

这么一来，就实现了笔者在本节初给出的那段 JavaScript 样例所能使用的扩展了。接下来一起验证一下吧。

```
$ node-gyp rebuild
...
$ node
> const addon = require("./build/Release/download");
undefined
> addon.download("http://www.csdn.net/", function processor(data) { console.
log(data); }, function callback() { console.log("done"); });
undefined
# 过了一会儿后
<Buffer 3c ... >
<Buffer 61 ... >
<Buffer 6e ... >
<Buffer 87 ... >
<Buffer 22 ... >
...
done
```

由于随书代码中的本样例代码并没有加入 HTTPS 支持，因此在测试的时候请使用 HTTP 协议的链接。如果大家有兴趣，可以参照我们先前使用过的 `minihttp` 的一些样例，尤其是它们的 `binding.gyp` 文件，将该样例改造成支持 HTTPS 的扩展。

可以看到，在我们下载 CSDN¹ 首页的时候，`Buffer` 分了好几拨进入我们声明的 `processor` 函数，我们在函数中将其输出；并在 `callback` 被执行的时候输出了一个 `done` 表示整个请求结束了。

¹ <http://www.csdn.net/>

大家也可以尝试一下，将 `processor` 和 `callback` 函数修改一下：

```
let content = "";
function processor(data) {
    content += data.toString();
}

function callback() {
    console.log(content);
}
```

当使用了这样的函数后，再执行 `addon.download()` 会发现，最后在 `callback` 中输出了整个 CSDN 首页的 HTML 源码。

注意本样例没有对各种异常情况做处理。有兴趣的读者可以尝试着通过 `TryCatch` 以及 `SetErrorMessage()` 等方式给这个 `download()` 函数加上一些异常处理。

5.5.5 小结

本节内容让大家初步尝到了在 Node.js 的 C++ 扩展中使用异步的“甜头”。在完全不会使用 `libuv` 的情况下，通过继承 `AsyncWorker` 或者 `AsyncProgressWorker` 以及结合 `AsyncQueueWorker()` 函数就能轻轻松松地写出一段异步的代码。

但是如果想要更深一步地使用异步代码，还请大家移步到第 6 章阅读 `libuv` 的相关内容。

5.5.6 参考资料

- [1] `nan/asyncworker.md` - Asynchronous worker helpers: <https://github.com/nodejs/nan/blob/master/doc/asyncworker.md>.
- [2] `nan/callback.md` - `Nan::Callback`: <https://github.com/nodejs/nan/blob/master/doc/callback.md>.
- [3] Thread pool work scheduling — libuv documentation: <http://docs.libuv.org/en/v1.x/threadpool.html>.

6

异步之旅——libuv

在第 5 章中，我们初次用 NAN 尝到了 Node.js 的 C++ 扩展异步编程的“甜头”。在本章中笔者将详细介绍如何用 libuv 进行异步编程。图 6-1 为 libuv 的 Logo。



图 6-1 libuv 的 Logo

libuv 一开始就是为了 Node.js 而生的，后来才逐渐被其他一些程序所用，例如 Luvit¹、Julia²、pyuv³、NeoVIM⁴ 等。

libuv 是一个高性能的、事件驱动 I/O 的跨平台 API 集。⁵

1 异步 I/O 版本的 Lua: <https://luvit.io/>。

2 Julia 语言，一门用于数值计算的高性能动态编程语言: <https://julialang.org/>。

3 libuv 的 Python 版 API: <https://pyuv.readthedocs.io>。

4 VIM 的一个重构版本: <https://neovim.io>。

5 翻译自 uvbook。

一开始的时候，Node.js 本来使用 Chrome V8 和 Marc Lehmann 写的 libev¹ 进行整合。后来人们发现，Node.js 越来越流行了，使之跨平台（尤指 Windows 平台）势在必行，而问题在于 libev 并不支持 Windows。于是 libuv 出现了，一开始它是基于 libev 进行开发的；而在 Windows 下则是基于 IOCP² 的，IOCP 类似于 kqueue 或者 epoll³。

再到后来的 libuv 版本（即 Node.js 0.9 所使用的 libuv 版本），直接就移除了对 libev 的依赖，完全由 libuv 自身实现对底层 I/O 模型的调用。

6.1 基础概念

众所周知，在 Node.js 中的很多地方都是强制异步的，至少在早期版本中很多功能均没有 *Sync() 类的 API（如 readdirSync() 等）。而在 Node.js 的异步中，事件循环充当着主体的角色，而事件循环就是由 libuv 提供的。此外，libuv 还提供了类似于计时器（timer）、非阻塞的网络操作、异步文件系统操作、子进程等特性。

总而言之，libuv 使用了事件驱动、异步 I/O 的模型。它提供了句柄（Handle）和流（Stream）为套接字（Socket），其他一些 I/O 实体提供了高级的抽象，此外还有跨平台的文件 I/O 和线程的功能。

笔者可以使用一张描述 libuv 不同子模块关系的图来描述它的结构，如图 6-2 所示。

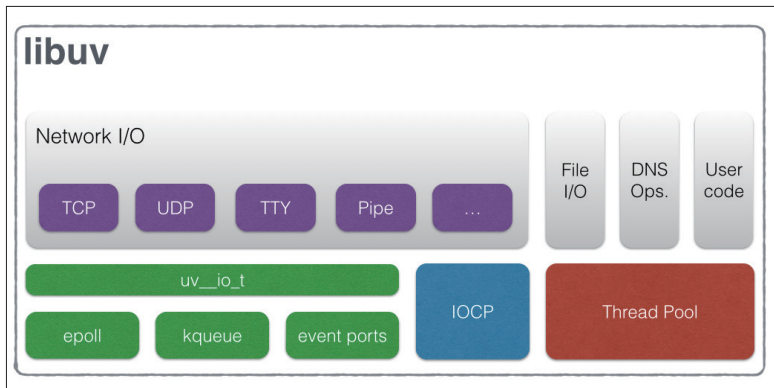


图 6-2 libuv 结构图⁴

¹ 也是一个事件驱动 I/O 的 API 集（库）：<http://software.schmorp.de/pkg/libev.html>。

² 即 I/O 完成端口（I/O Completion Port），是 Windows 下的异步 I/O API，它可以高效地将 I/O 事件通知给应用程序。

³ kqueue 对应 FreeBSD，epoll 对应 Linux。

⁴ 图片来自 libuv 设计概观：<http://docs.libuv.org/en/v1.x/design.html>。

6.1.1 事件循环

除了科学计算类的项目和重 CPU 操作的项目外，我们日常面对开发的其余项目一般是偏 I/O 处理的。熟悉 C 或者 C++ 的读者应该对 `read`、`fprintf` 等函数不会感到陌生，不过这些都是阻塞型的函数，使用它们读取、输出文件的时候会阻塞在当前线程。也就是说，阻塞时的效率就比较看重硬盘的读 / 写速度了。除此之外，读 / 写网络请求有可能性能更低，在读 / 写完成之前，你的程序（除非是自行实现多线程）就“呆若木鸡”，无法执行别的逻辑了。

1. 事件循环的基石

为了解决程序被 I/O 操作阻塞的局面，市面上最常见的做法就是使用多线程。我们为各阻塞 I/O 的操作都启动一个常规的子线程（或者是从一个线程池中取）。当阻塞操作执行完毕之后，程序将通知另一个线程。不过，这种方法非常耗费 CPU 资源。

并且，在多线程的操作中，我们通常还要搭配上锁一起使用。线程与锁模型就像一辆福特 T 型车：驾驶它可以到达目的地，但与新的技术相比，它显得原始和难以驾驭，且不可靠，另外还有点儿危险。¹

相比之下，libuv 的解决方案就封装得比较深了——异步、非阻塞。基本上所有的现代操作系统都提供了事件通知的机制。

例如，针对一个 Socket 调用的常规 `read()` 函数就会阻塞住线程，直到它真的接收到内容。但是，换一种姿态的话，我们可以在程序里面请求操作系统来监视 Socket，然后在队列中放置一个事件通知。这样一来，你的程序就可以方便地检视事件并且获取到你想要的数据了，这就是我们老生常谈的 IOCP、kqueue 或者 epoll 了。这些机制是异步的，因为程序只需要发起一个操作，然后在另一个时间接收结果，而在这之间，程序可以无障碍地执行其他的任务。

如果我们使用早期的 `select()` 机制，那使用的就是“鬼子进村”的策略。即一遍遍的询问：“鬼子进村了吗？”“鬼子进村了吗？”……大量的 CPU 时间都耗了进去。

而使用 IOCP 等，则变成了派一些个人去站岗，鬼子来了就可以得到通知，效率自然高了许多。

——整理自云风的 BLOG《IOCP, kqueue, epoll……有多重要？》²

这种机制是支撑起 libuv 事件循环的基石。

¹ 摘自《七周七并发模型》。

² http://blog.codingnow.com/2006/04/iocp_kqueue_epoll.html

2. Node.js 举例

在事件驱动的 Node.js 中，在一些操作的时候监听一些特定的事件，并在事件发生的时候做出响应，就如这样的一段代码：

```
var http = require("http");

http.createServer(function(req, resp) {
    resp.writeHead(200, { "Content-Type": "text/html" });
    resp.write("<h1> 来一打 C++ 扩展</h1>");
    resp.end("<p> 不然蛋花汤咬你哦 ^|·▽·| / *~ ● </p>");
}).listen(3000);

console.log("Http Server is listening at port: 3000");
```

明眼人一眼就看出来了，代码通过 Node.js 被执行之后，我们在浏览器中打开 `http://127.0.0.1:3000` 之后，就会出现如图 6-3 所示的结果。



图 6-3 HTTP Demo 页面

学过 Node.js 的人应该都知道，被传入 `createServer()` 中的匿名函数就是用于处理“有请求进来”这个事件的回调函数。当有请求进来时，这个函数就会被放入事件循环中执行。

对于参数、变量的命名，不推荐盲目从众。例如 `res` 的语义应与 `ret` 类似，都表示 `result` 的缩写；而对于响应（`Response`）的变量，推荐使用 `resp` 来代表。以上只是笔者的主观建议，仅供参考。

3. 流程介绍

将事件循环作为流程图画出来，就会如图 6-4 所示。

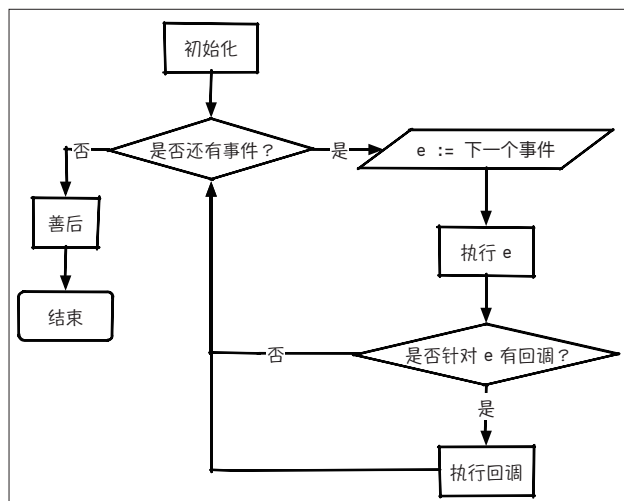


图 6-4 Event Loop 流程图

对于流程图中的“事件”一词，这里有几种样例可供参考：

- 文件已准备好，处于可写状态了；
- 一个 Socket 已有数据，处于可读状态了；
- 一个计时器超时了。

注意：即使在 libuv 中调用一些底层的 I/O 操作，这些逻辑也会在后台执行，我们并不需要关心。计算机硬件的工作原理决定了线程作为基本的处理器单元，libuv 和操作系统实际上通常也会运行一些后台的工作线程或者轮询，并以非阻塞的方式执行任务。但是这些线程的具体概念也并不需要我们关心。甚至在 Node.js 中，暴露给 JavaScript 开发者的只有主事件循环的线程，其他所有底层的异步工作均在相对于开发者的黑盒中完成。

事件循环由 libuv 的 `uv_run()` 函数封装，它是在使用 libuv 时所有功能的最终函数。

4. 默认事件循环

当你在自己的程序中只使用一个事件循环的话，可以使用 `uv_default_loop()` 来获取 libuv 默认创建的事件循环句柄。

事实上，Node.js 就使用了 libuv 中的默认循环作为它的主事件循环。

5. `uv_run()`

`uv_run()` 这个函数顾名思义，就是将某个事件循环开始运转起来。它的原型如下：

```
int uv_run(uv_loop_t* loop, uv_run_mode mode)
```

其中 `uv_run_mode` 是一个枚举类型，有以下几种值：

```
typedef enum {
    UV_RUN_DEFAULT = 0,
    UV_RUN_ONCE,
    UV_RUN_NOWAIT
} uv_run_mode;
```

不过在 Node.js 的原生扩展开发中，这些都不是我们需要关心的，因为 Node.js 事先已经把各种初始化步骤做完了。有兴趣的读者可自行阅读 libuv 的官方文档。

6.1.2 句柄（Handle）与请求（Request）

1. 句柄（Handle）

libuv 的一个基本单位就是句柄。事件的操作都是通过创建 I/O 设备、计时器或者进程等句柄来进行的。句柄在 libuv 中有非常多的种类，并且以 `uv_句柄类型_t` 来命名类型。对于一个 libuv 的句柄，我们通常要为其设置一个对应的回调函数。

- `uv_loop_t`：事件循环句柄。
- `uv_handle_t`：所有句柄类型的基础类型。
- `uv_stream_t`：双工通信通道的抽象句柄。
- `uv_tcp_t`：TCP 句柄，`uv_stream_t` 的子类型。
- `uv_udp_t`：UDP 句柄。
- `uv_pipe_t`：UNIX 本地域套接字或者 Windows 上的命名管道的句柄，`uv_stream_t` 的子类型。
- `uv_tty_t`：终端的 Stream 句柄。
- `uv_poll_t`：用于监视文件描述符的句柄，其类似于 `poll()`¹。
- `uv_timer_t`：定时任务（计时器）的句柄。
- `uv_prepare_t`：Prepare 阶段句柄，该句柄对应的回调函数会在事件循环的每次循环迭代初执行一次，即在轮询 I/O 之前。

¹ <https://linux.die.net/man/2/poll>

- `uv_check_t`: Check 阶段句柄, 该句柄对应的回调函数会在事件循环的每次循环迭代末执行一次, 即在轮询 I/O 之后。
- `uv_idle_t`: 空转句柄, 该句柄对应的回调函数会在事件循环的每次循环迭代初执行一次, 比 Prepare 阶段还要早。
- `uv_async_t`: 异步句柄, 允许用户从另一个非 libuv 维护的线程中唤醒 libuv 事件循环并执行该句柄对应的回调函数。
- `uv_process_t`: 子进程句柄, libuv 会创建一个新的进程给开发者的逻辑代码控制, 并且提供了基于 Stream 的主子进程通信频道。
- `uv_fs_event_t`: 文件系统监视句柄, 用于如某个文件路径对应的内容是否改变, 如某文件是否被重命名, 等等。
- `uv_fs_poll_t`: 文件系统监视句柄, 其类似于 `uv_fs_event_t`。不同的是, `uv_fs_poll_t` 使用 `stat` 来监测文件系统的变动。
- `uv_signal_t`: 信号句柄, 其在每个事件循环的基础上实现 UNIX 风格的信号处理, 哪怕是在 Windows 下, libuv 也模拟了一些信号, 如 `SIGINT`、`SIGBREAK` 等。

通常, 我们的句柄都可以通过这种方式来初始化:

```
uv_句柄类型_init(uv_loop_t *, uv_句柄类型_t *)
```

例如:

```
uv_async_t async;
uv_async_init(uv_default_loop(), &async);
```

2. 请求 (Request)

libuv 中的句柄代表了一个长生命周期的抽象“物件”。在相应的句柄上, 有很多与之关联的请求。与句柄不同的是, 请求的生命周期短 (通常可被理解为一性次的), 如对应句柄上的一次 I/O 操作。请求用于存储单次操作的启动阶段与回调阶段的上下文数据。

例如, 一个 UDP 套接字在 libuv 中使用 `uv_udp_t` 这个句柄来代表, 而每一次往该套接字写入数据之后, libuv 都会往回调函数中传递一个 `uv_udp_send_t` 请求。

在 libuv 中, 有以下这些请求的类型。

- `uv_req_t`: 所有请求类型的基础类型。
- `uv_getaddrinfo_t`: 用于 `uv_getaddrinfo()`。

- `uv_getnameinfo_t`: 用于 `uv_getnameinfo()`。
- `uv_shutdown_t`: 用于 `uv_shutdown()`。
- `uv_write_t`: 用于 `uv_write()` 等。
- `uv_connect_t`: 用于 `uv_listen()`。
- `uv_udp_send_t`: 用于 `uv_udp_send()`。
- `uv_fs_t`: 用于 `uv_fs_req_cleanup()` 等。
- `uv_work_t`: 用于 `uv_queue_work()`。

3. 回调函数 (Callback)

在 libuv 中回调函数的定义是指那些 libuv 所关注的事件发生后，被调用的函数。事件发生后，业务级的逻辑应当被开发者在回调函数中实现。

例如，一个文件系统相关句柄的回调函数会接收到从文件中读取到的内容，一个定时器的回调函数会在超时后被触发，等等。

6.1.3 尝尝甜头

1. 最简单的程序

跟着笔者先写一个最简单的使用 libuv 的程序来尝尝甜头吧。新建一个 `main.c` 的文件，写入如下的代码（前提是你已经能正常引入 libuv，例如在 Ubuntu 下通过 `$ [sudo] apt-get install libuv libuv-dev` 来安装）：

```
#include <stdio.h>
#include <stdlib.h>
#include <uv.h>

int main()
{
    uv_loop_t* loop = (loop*)malloc(sizeof(uv_loop_t));
    uv_loop_init(loop);

    printf("Now quitting.\n");
    uv_run(loop, UV_RUN_DEFAULT);

    uv_loop_close(loop);
    free(loop);
    return 0;
}
```

这段代码会在程序启动的时候马上启动一个事件循环，并且在启动后又马上退出。

为什么会马上退出呢？对照着图 6-4 来看，很容易能发现，是因为其循环中并没有任何事件等待执行，所以很快就跳脱出了循环的流程。

2. Node.js 中的 libuv 体现

让我们将目光转向 Node.js 6.9.4 版本中 `src/node.cc` 的第 4611 ~ 4629 行。

```
{
  SealHandleScope seal(isolate);
  bool more;
  do {
    v8_platform.PumpMessageLoop(isolate);
    more = uv_run(env->event_loop(), UV_RUN_ONCE);

    if (more == false) {
      v8_platform.PumpMessageLoop(isolate);
      EmitBeforeExit(env);

      // 如果循环在发生事件之后或者在运行一些回调之后生效，则触发 `beforeExit`
      more = uv_loop_alive(env->event_loop());
      if (uv_run(env->event_loop(), UV_RUN_NOWAIT) != 0)
        more = true;
    }
  } while (more == true);
}
```

这段就是 Node.js 中 libuv 主事件循环相关的代码了。其间也能隐约看出笔者先前流程图中的一点点影子。

在循环中的第一步是先执行 Chrome V8 的消息循环。`v8_platform.PlupMessageLoop()` 实际上就是对 `v8::platform::PumpMessageLoop()` 的一个封装。`v8::platform` 是一个多线程的模型，因为其中可能会有多个 Isolate 实例。不过在 Node.js 中，只有一个 Isolate。线程以 Isolate 实例的指针作为键名保存在一个集合中。在执行消息循环时，在当前的线程中只需要取出当前 Isolate 的队列即可，该队列中有一些定时器的任务。如果是在 Chrome 下，它们有可能就是浏览器环境下的一些延时操作。定时器以 `pair<double, Task*>` 的结构维护一棵红黑树，便于查找到最早的定时器以执行任务。

在之后就是笔者在 6.1.1 节提到过的 `uv_run()` 函数了。`uv_run()` 就是事件循环的入口，从这里开始执行事先定义的循环。执行完之后会判断一下是否还有事件等待执行，若没有了则再取一下 Chrome V8 中的消息，然后触发 `EmitBeforeExit()`。在此会有一个

`node::MakeCallback()` 的调用。如果在此处执行的代码中有类似数据库访问等异步操作，那么 `uv_run()` 中事件循环就又“活”过来了，于是 `more` 又会等于 `true`。

关于 libuv 在 Node.js 中的体现到这里就结束了。因为若没有后续的一些储备，大家理解起来会有些吃力。

3. 空转 (Idling)

在本书中，大家只需要知道一个空转句柄会在事件循环的每次循环之初被执行即可。

为了让大家能对该实例更亲切些，笔者将本实例融入 Node.js 原生扩展中。

请大家打开随书代码的“30. libuv idle”中的 `idle.cc` 吧。

```
unsigned int idle_times = 0;
uv_idle_t* idle = 0;

NAN_METHOD(StartIdle)
{
    if(idle != 0)
    {
        return Nan::ThrowError(" 上一个空转仍在进行。");
    }

    idle_times = Nan::To<unsigned int>(info[0]).FromJust();

    idle = new uv_idle_t();

    // 初始化空转句柄
    uv_idle_init(uv_default_loop(), idle);

    // 将 `test_idle` 函数绑定给空转句柄并开始
    // 即每个空转阶段都会执行 `test_idle` 函数
    uv_idle_start(idle, test_idle);
}
```

在导出的 `startIdle` 函数中，我们先判断全局的 `idle` 是否存在，若存在则抛错；若不存在，我们将获取传入的空转执行次数，然后生成一个全局的 `idle` 句柄并通过 `uv_idle_init()` 函数初始化它。做好了这些事情之后，我们将 `test_idle()` 这个函数绑定到空转句柄上，即在每个空转阶段都会执行 `test_idle()` 这个函数。

下面再来看看 `test_idle()` 这个空转回调函数吧。

```

void test_idle(uv_idle_t* handle)
{
    // 空转次数减一
    idle_times--;

    // 输出一点内容
    printf(" 还剩 %u 次空转。\\n", idle_times);

    // 若空转次数到达，则该空转停止
    if(idle_times == 0)
    {
        idle = 0;

        // 空转停止
        uv_idle_stop(handle);

        // 释放空转句柄
        delete handle;
    }
}

```

回调一开始，就将全局的空转次数减一，并且输出一句我们肉眼能见的终端语句。最后判断空转次数是否已经到了，若到了则将全局的空转句柄赋零，然后通过 `uv_idle_stop()` 函数使该空转句柄停止并释放它。

一个简单的空转示例就完成了。大家可以在终端中执行以下这段代码试试看。

```

$ node-gyp rebuild
...
$ node
> const addon = require("../build/Release/idle");
undefined
> addon.startIdle(5);
undefined
还剩 4 次空转。
还剩 3 次空转。
还剩 2 次空转。
还剩 1 次空转。
还剩 0 次空转。

```

如上所示，在我们执行 `startIdle()` 函数后，在接下来的事件循环中的 5 次循环各自的空转阶段，都输出了我们所期待的内容。

可能大家认为效果不明显，为此笔者还提供了—个 `test.js` 文件供大家参考。

```
const addon = require("../build/Release/idle");

addon.startIdle(100);

try {
    addon.startIdle(1);
} catch(e) {
    console.error(e);
}

let times = 100;
function tick() {
    times--;
    console.log(`还剩 ${times} 次 tick。`);

    if(times === 0) return;

    setImmediate(tick);
}

setImmediate(tick);
```

在这期间，我们先开始一个 100 次的空转，并且在当前的 tick 中又同时欲开启一个 1 次的空转。当然后者会抛错说已经有一个空转正在执行了，所以笔者通过 `try...catch` 将其包起来。再接下来笔者通过 `setImmediate()` 函数来执行 100 次 tick，每次都输出一个 `还剩 \${times} 次 tick。` 的内容。

执行 `$ node test` 后的结果会这么输出：

```
Error: 上一个空转仍在进行。
    at Error (native)
    at Object.<anonymous> (/Users/USER/30. libuv idle/test.js:13:11)
    at Module._compile (module.js:570:32)
    at Object.Module._extensions..js (module.js:579:10)
    at Module.load (module.js:487:32)
    at tryModuleLoad (module.js:446:12)
    at Function.Module._load (module.js:438:3)
    at Module.runMain (module.js:604:10)
    at run (bootstrap_node.js:394:7)
    at startup (bootstrap_node.js:149:9)
还剩 99 次空转。
还剩 99 次 tick。
还剩 98 次空转。
还剩 98 次 tick。
...
```

```
还剩 0 次空转。  
还剩 0 次 tick。
```

通过上面的输出结果我们能看到，在 100 个 tick 中，空转的回调函数总是先于 `setImmediate()` 所设置的函数执行，并且在一开始也如愿以偿地抛出了一个错误。

6.1.4 小结

在本节中，笔者介绍了 libuv 中的一些最基本概念：

- 事件循环（Event Loop）；
- 句柄（Handle）；
- 请求（Request）。

前面通过两个范例让大家对使用 libuv 进行 Node.js 下的原生扩展开发有了一个初步的了解。

不过本节对于这些基本概念并不深入，刚好够用于 Node.js 的原生扩展开发。

在本书的后续章节中，会着重介绍关于跨线程操作的编程方式，但会省略文件系统、网络、进程等相关的内容。毕竟一旦读者使用 C++ 来写原生扩展，通常情况下要么是为了 CPU 计算的效率，要么是接外部已有的代码或库。若是为了 CPU 计算的效率，一般是不会有文件系统等操作的；而如果是后者的话，则通常读者要接的库已经在自己的线程中封装好了它特有的网络操作、进程操作和文件系统操作等内容了。

如果读者有兴趣，可以自行翻阅 libuv 的文档以及 *An Introduction to libuv*¹ 这本电子书。

6.1.5 参考资料

- [1] Paul Butcher, 黄炎. 七周七并发模型 [M]. 北京：人民邮电出版社，2015.
- [2] Basics of libuv: <http://nikhilm.github.io/uvbook/basics.html>.
- [3] 神之门 V8——Event loop 的舞池盛宴 (2): <http://blog.csdn.net/u013700510/article/details/53401838>.

¹ <http://nikhilm.github.io/uvbook>

6.2 libuv 的跨线程编程基础

很多人如果是“空降”到此的，乍一看章节的标题可能会觉得很奇怪，在很多人的传统认知里，libuv 和 Node.js 是单线程的。实际上是因为真正多线程的操作已经被封装了——对于上层开发者来说，不涉及底层内容的话，他们的开发方式的确都是单线程的。

但是实际上在 libuv 底层也还是多线程的。正如笔者先前所说的一样，在 Node.js 或者 libuv 的主事件循环的线程上一直运行着用户的业务逻辑，而其他线程则负责执行各种异步操作。这看起来有点像图 6-5。

正如笔者开发的阿里云 ONS SDK 一样，官方提供的 C++ SDK 封装了一套网络请求的机制与其消息队列进行交互，并且所有的消息读取逻辑都是在它们的线程中完成的。笔者要做的就是将其线程中的逻辑与 Node.js 的主事件循环串联起来，正如图 6-5 中的“人家的 SDK”那附近的一小块象形逻辑一样。

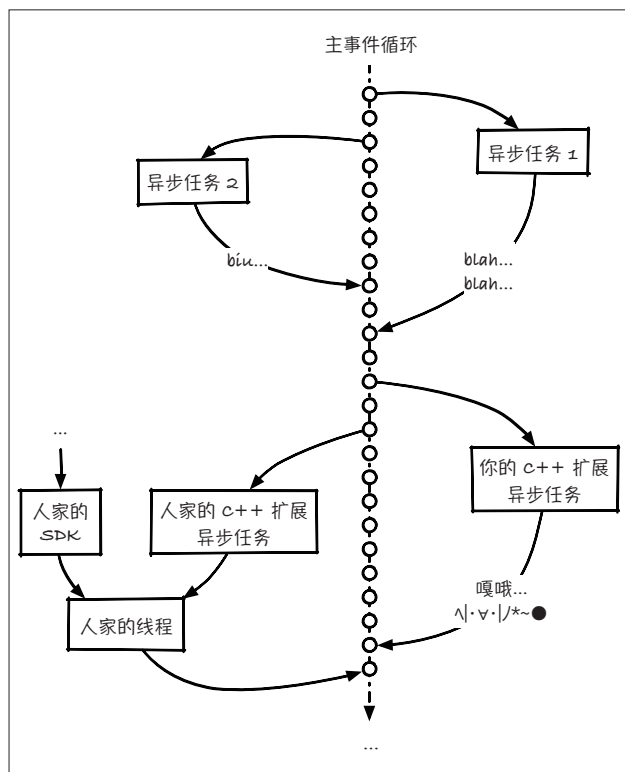


图 6-5 事件循环象形图

由于 libuv 的主线程与外部其他线程无法直接交互（会有各种各样的“坑”），因此就有了本节内容——libuv 的跨线程编程。

6.2.1 libuv 的线程

线程与锁模型非常简单直接，几乎所有编程语言都以某种形式对其提供了支持，且不对其使用方式加以限制。换句话说，对于不精通该模型的程序员，编程语言层面没有提供足够的帮助，这使得程序容易出错且难以维护。

libuv 这个 C 语言库自然也对线程提供了封装。它的线程允许你的程序在一个它的线程中异步执行你的阻塞逻辑，直到拿到结果后将通知返还给主事件循环。它是基于 POSIX¹ 的 pthread² 以及 Windows 下的 thread 来实现的。

有一点需要注意的是，其他的 libuv 操作都是可预知的，**即都是依赖于事件循环和回调的原则**。但是 libuv 的线程是不可预知的，甚至可以完全脱离 libuv 的事件循环独立运行，它们在需要的时候有可能会被阻塞，通过返回值来产生信号错误等。

不过考虑到不同平台上的原生线程 API（包括语法和语义）都不相似，于是乎 libuv 对线程支持的 API 也很有限——主要是众口难调。

1. 主要 API

在 libuv 的线程 API 中所需要的句柄类型是 `uv_thread_t`。然后最主要用到的两个 API 分别如下：

- `uv_thread_create(uv_thread_t*, uv_thread_cb, void*)`;
- `uv_thread_join(uv_thread_t)`。

其中 `uv_thread_create()` 函数用于创建一个线程，将得到的句柄赋值给传入的句柄指针。

`uv_thread_cb` 就是线程处理函数了。它的返回值是 `void` 并且接收的参数是一个 `void*` 的指针，即通过 `uv_thread_create()` 最后一个参数传入的值。

下面举个例子，这就是一个合法的 `uv_thread_cb`：

```
void func(void* arg)
{
    int times = *((int*)arg);
    while(times--)
```

¹ UNIX 下的可移植操作系统接口（Portable Operating System Interface of UNIX）。

² <http://man7.org/linux/man-pages/man7/pthreads.7.html>

```

{
    sleep(1);
    printf("output\n");
}
}

```

而 `uv_thread_join()` 函数的意义就是等待传入的线程句柄所对应的线程运行结束——即阻塞当前线程，直到原线程结束。

2. 用于演示的睡眠排序

我们都知道，在 Node.js 中如果你要实现 `sleep` 等类似操作，只能通过类似于 `setTimeout()` 等方式进行，而且它们是非阻塞的。现在我们能通过 `libuv` 使用线程相关函数，创建新的线程并阻塞住 Node.js 的主事件循环。虽然这么做并没有什么实际意义，但是本节这么做主要是为了演示这些 API。

睡眠排序¹也是一个非常有意思的算法，尽管其没有太大的实际意义，但是它有助于开发大家的“脑洞”。

通常，睡眠排序会对每个待排序元素启动一个定时任务，该任务定的时长是该元素的权重（如对数字排序，则时长可以是数字本身）。一旦定时任务被触发，则将该元素放入结果数组中。这么一来，当所有的任务执行完毕之后，数组也就排序完成了。

例如，用 JavaScript 来实现一个睡眠排序的话，就可以这样写：

```

const ret = [];
[ 2, 4, 6, 9, 1 ].forEach(num => {
    setTimeout(() => ret.push(num), num * 1000);
});

```

它的运行结果就是 9 秒后 `ret` 被排序完成，结果是 `[1, 2, 4, 6, 9]`。

接下来笔者解析如何用 `libuv` 来实现一个阻塞版的睡眠排序。请大家打开随书代码“31. `libuv sleep sort`”中的 `sleep.cc` 学习一下吧。

下面先来解析一下主要的导出函数。

```

struct ThreadArg {
    std::vector<uint32_t>* vec;
    uint32_t num;
};

```

¹ https://rosettacode.org/wiki/Sorting_algorithms/Sleep_sort

```

NAN_METHOD(Sort)
{
    if(info.Length() < 1 || !info[0]->IsArray())
    {
        return Nan::ThrowError("Wrong argument");
    }

    Local<Array> array = info[0].As<Array>();
    if(!array->Length())
    {
        return info.GetReturnValue().Set(Nan::New<Array>());
    }

    std::vector<uint32_t> orig;
    for(uint32_t i = 0; i < array->Length(); i++)
    {
        if(!Nan::Get(array, i).ToLocalChecked()->IsUint32())
        {
            return Nan::ThrowTypeError("Elements should be unsigned int.");
        }

        orig.push_back(Nan::To<uint32_t>(Nan::Get(array,
i).ToLocalChecked()).FromJust());
    }

    std::vector<uint32_t> vec;
    std::vector<uv_thread_t> handles(array->Length());
    for(uint32_t i = 0; i < array->Length(); i++)
    {
        ThreadArg* arg = new ThreadArg();
        arg->vec = &vec;
        arg->num = orig[i];
        uv_thread_create(&handles[i], Sleep, arg);
    }

    for(uint32_t i = 0; i < handles.size(); i++)
    {
        uv_thread_join(&handles[i]);
    }

    Local<Array> ret = Nan::New<Array>();
    for(uint32_t i = 0; i < array->Length(); i++)
    {
        Local<Value> v = Nan::New(vec[i]);
        Nan::Set(ret, i, v);
    }
}

```



```

    }

    info.GetReturnValue().Set(ret);
}

```

一开始都是一些参数以及参数类型等的校验——该函数只接收一个里面所有元素都是无符号整型的数组，否则会抛出错误。

然后通过一个 for 循环，将数组内的元素转成 C++ 原生的 `uint32_t` 并存储到一个 `std::vector<uint32_t>` 中。而后第二个 for 循环就步入了睡眠排序的正轨。

```

std::vector<uint32_t> vec;
std::vector<uv_thread_t> handles(array->Length());
for(uint32_t i = 0; i < array->Length(); i++)
{
    ThreadArg* arg = new ThreadArg();
    arg->vec = &vec;
    arg->num = orig[i];
    uv_thread_create(&handles[i], Sleep, arg);
}

```

这里首先生成一个新的 `ThreadArg` 对象，然后将结果数组 `vec` 和当前数值 `num` 进行赋值，并通过 `uv_thread_create()` 创建一个针对当前元素睡眠排序用的线程，它所对应的函数就是 `Sleep(void*)`。

在第 3 个 for 循环中，通过 `uv_thread_join()` 来等待每一个线程结束——也就是笔者所说的阻塞住当前线程（在这里等同于 Node.js 主事件循环）。

等到阻塞结束后，我们再通过一个 for 循环中的 `Nan::Set()` 将结果设置到我们即将返回的 V8 数组对象中。最后一句代码就是返回这个排序后的结果了。

讲完了这个导出函数后，我们可以看看线程函数中到底是怎样的。

```

void _Sleep(int sleep_ms)
{
#ifdef WINDOWS
    Sleep(sleep_ms);
#else
    usleep(sleep_ms * 1000);
#endif
}

void Sleep(void* _)
{

```

```

ThreadArg* arg = (ThreadArg*)_;

// 睡上 num * 100 毫秒
_sleep(arg->num * 100);
arg->vec->push_back(arg->num);

delete arg;
}

```

上面代码中的 `Sleep()` 就是线程处理函数了。在函数中，笔者先把传进来的参数给转换回我们需要的 `ThreadArg*` 类型。然后通过 `_Sleep()` 函数“睡”上数值乘以 100 毫秒的时间（`_Sleep()` 函数通过宏判断来跨平台，Windows 平台使用 `Sleep()`¹ 函数，而其他平台使用 `usleep()`² 函数）。在睡眠结束后，往传入的 `vec` 向量中推入当前线程所对应的数值本体。最后回收一下 `arg` 内存，这个线程就结束了。

这样一来，如果是 `[3, 1, 2]` 这么一个数组的话，当前 C++ 模块就会开启 3 个线程，分别睡眠 300、100 和 200 毫秒，然后分别将 1、2、3 推入新的数组中。其在主事件循环中会被阻塞，直到所有线程结束（约 300 毫秒），再返回结果。

在阻塞过程中，由于是在 Node.js 的主事件循环中，因此所有其他在 JavaScript 层面以及在主事件循环操作的 C++ 层面的逻辑都不会被执行，直到阻塞结束。这就是笔者在本节之初就强调本样例在线上项目中并没有什么实际价值的原因。

代码解析讲完了，现在开始试试看吧。

```

$ node-gyp rebuild
...
$ node
> const addon = require("./build/Release/sleep");
undefined
> addon.sort([3,2,1.1]);
TypeError: Elements should be unsigned int.
    at TypeError (native)
    at repl:1:7
    at sigintHandlersWrap (vm.js:96:12)
    at REPLServer.defaultEval (repl.js:313:29)
    at bound (domain.js:280:14)
    at REPLServer.<anonymous> (repl.js:513:10)
    at emitOne (events.js:101:20)
    at REPLServer.emit (events.js:188:7)

```

1 即 `VOID WINAPI Sleep(DWORD dwMilliseconds)`，[https://msdn.microsoft.com/zh-cn/library/windows/desktop/ms686298\(v=vs.85\).aspx](https://msdn.microsoft.com/zh-cn/library/windows/desktop/ms686298(v=vs.85).aspx)。

2 即 `int usleep(useconds_t usec)`，<https://linux.die.net/man/3/usleep>。

```

    at REPLServer.Interface._onLine (readline.js:239:10)
    at REPLServer.Interface._line (readline.js:585:8)
> addon.sort([3,2,1,4]);
[ 1, 2, 3, 4 ] # 阻塞 400 毫秒

```

一切看起来都如你所愿，让我们再来试试看吧。

```

$ node
> const addon = require("../build/Release/sleep");
undefined
> addon.sort([3,2,1,3]);
# 有一定概率会崩溃
[1]      82919 segmentation fault  node

```

大家会发现，当数组内存在两个以上的相同数字的时候，多试几次，总有一定概率会导致整个程序崩溃。为什么会这样呢？

因为其跟所有其他的 STL¹ 容器一样，都不是线程安全式的。一旦两个线程同时操作同一个容器（在本样例中就是操作同一个 vec 对象），很容易就发生崩溃。所以这段样例代码实际上是不完整的，或者说的不安全，至少在有多个相同数字时，由于不同的线程会睡眠同样的毫秒数，因此会导致它们同时操作一个向量容器，以致崩溃。

6.2.2 同步原语（Synchronization Primitive）

事先声明一下，这里讲的同步并不是我们在 Node.js 中通常所说的同步概念。

避免对同一数据的并发访问（通常由中断、对称多处理器、内核抢占等引起）被称为同步²。同步原语就是用来执行同步的一些实体。

6.2.1 节的最后笔者留了一个悬念，那就是多线程同时操作一个“物件”的时候，很容易形成资源抢占的问题，导致进程崩溃。本节中介绍的同步原语就能比较好地解决这些问题。

本节中笔者所讲的同步原语指的是 libuv 中的一些同步原语，有以下几种：

- 互斥锁（Mutex Lock）；
- 读 / 写锁（Write-Read Lock）；
- 信号量（Semaphore）；
- 条件变量（Condition Variable）；
- 屏障（Barrier）。

¹ Standard Template Library, https://en.wikipedia.org/wiki/Standard_Template_Library

² http://blog.csdn.net/npv_lp/article/details/7262388

1. 互斥锁 (Mutex Lock)

互斥 (Mutual exclusion, 缩写为 Mutex) 锁是一种用于多线程编程中, 防止两个线程同时对同一公共资源 (比如全局变量) 进行读 / 写的机制。该目的通过将代码切片成一个一个的临界区域 (critical section) 来达成。

libuv 中的互斥锁几乎等同于 pthread 中的互斥锁¹。如果大家对 Windows 下的多线程编程有了解的话, 也可以类比一下 CRITICAL_SECTION²。

互斥锁的句柄类型是 uv_mutex_t, 它有以下 5 个相关函数:

- int uv_mutex_init(uv_mutex_t* handle)。
- void uv_mutex_destroy(uv_mutex_t* handle)。
- void uv_mutex_lock(uv_mutex_t* handle)。
- int uv_mutex_trylock(uv_mutex_t* handle)。
- void uv_mutex_unlock(uv_mutex_t* handle)。

顾名思义, 这 5 个函数分别为初始化一个互斥句柄、释放一个互斥句柄、将一个互斥句柄锁住、尝试锁住一个互斥句柄以及解锁一个互斥句柄。

关于互斥锁, 我们可以尝试通过 C++ 对象的作用域生命周期来封装一个作用域互斥锁, 其原理有点类似于 V8 的句柄作用域。

```
struct ScopeLock {
    uv_mutex_t* handle;

    ScopeLock(uv_mutex_t* handle) : handle(handle)
    {
        uv_mutex_lock(handle);
    }

    ~ScopeLock()
    {
        uv_mutex_unlock(handle);
    }
};

int i;
uv_mutex_t handle; // 假设已在其他地方初始化
```

¹ http://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_mutex_lock.html

² [https://msdn.microsoft.com/zh-cn/library/windows/desktop/ms682530\(v=vs.85\).aspx](https://msdn.microsoft.com/zh-cn/library/windows/desktop/ms682530(v=vs.85).aspx)

```
void ScopeLockTest()
{
    ScopeLock lock(&handle);
    i = 0;
}
```

我们可以发现，当我们刚进入 `ScopeLockTest` 函数的时候，声明了一个 `lock` 对象，这个时候运行 `lock` 的构造函数，就锁住了 `handle` 互斥锁。而当 `ScopeLockTest` 函数运行完毕要退出这个函数的时候，`lock` 对象的生命周期也就走到了尽头；相对应地，它将会执行析构函数，那么就自然而然地解锁了。

其实无论 `ScopeLockTest` 这个函数怎么写，哪怕是中间有一些 `if` 分支条件判断直接通过 `return` 返回，只要是 `ScopeLockTest` 这个函数执行完毕，`lock` 就会自动析构，从而可达到解锁的过程。那么，不管是粗心还是细心的读者都不用为忘记退出临界区而烦恼了。¹

读者可以尝试着将前文讲述的睡眠排序通过互斥锁来修正一番。打开随书代码“32. libuv sleep sort advanced”中的 `mutex/sleep.cc` 阅读，或者跟着下文的步骤来处理先前的错误代码。

我们首先将刚才写的 `ScopeLock` 结构体复制到源码中，然后改造一下 `ThreadArg` 结构体。

```
struct ThreadArg {
    ...
    uv_mutex_t* mutex_handle;
};
```

我们在 `Sort` 函数中先初始化互斥锁句柄，并将其传入每个线程中。

```
...

std::vector<uint32_t> vec;
std::vector<uv_thread_t> handles(array->Length());
uv_mutex_t handle;
uv_mutex_init(&handle);
for(uint32_t i = 0; i < array->Length(); i++)
{
    ThreadArg* arg = new ThreadArg();
    ...
    arg->mutex_handle = &handle;
    uv_thread_create(&handles[i], Sleep, arg);
}
```

¹ 参考自 <https://xcoder.in/2012/09/08/scope-lock/>。

```
...
uv_mutex_destroy(&handle);
```

最后，在线程函数的合适位置中声明刚写好的作用域互斥锁即可。

```
void Sleep(void* _)
{
    ThreadArg* arg = (ThreadArg*)_;

    // 睡上 num * 100 毫秒
    _Sleep(arg->num * 100);

    ScopeLock lock(arg->mutex_handle);
    arg->vec->push_back(arg->num);

    delete arg;
}
```

大功告成后，大家可以再试试让之前的错误代码崩溃的数据。

```
$ node-gyp rebuild
...
$ node
> const addon = require("../build/Release/mutex_sleep");
undefined
> addon.sort([3,2,1,3]);
[ 1, 2, 3, 3 ] # 阻塞 300 毫秒左右
```

怎么样？不会崩溃了吧。因为笔者在线程中要操作同一个容器的时候，事先通过互斥锁将临界区锁起来了，这导致同时只有一个线程能继续走下去，直到解锁。

2. 读 / 写锁（Write-Read Lock）

与互斥锁相比，读 / 写锁是一个更细粒度的锁——其规则更加详细。

一个临界区可以同时被两个线程只读访问，也就是说只读锁能同时上锁。当一个临界区被只读锁锁住的时候，写锁要等其解锁才能继续。当一个临界区被写锁锁住的时候，其他线程相应的只读锁和写锁都无法上锁。

读 / 写锁的句柄类型为 `uv_rwlock_t`，与之相关的函数如下：

- `int uv_rwlock_init(uv_rwlock_t* rwlock);`
- `void uv_rwlock_destroy(uv_rwlock_t* rwlock);`

- `void uv_rwlock_rdlock(uv_rwlock_t* rwlock);`
- `int uv_rwlock_tryrdlock(uv_rwlock_t* rwlock);`
- `void uv_rwlock_rdunlock(uv_rwlock_t* rwlock);`
- `void uv_rwlock_wrlock(uv_rwlock_t* rwlock);`
- `int uv_rwlock_trywrlock(uv_rwlock_t* rwlock);`
- `void uv_rwlock_wrunlock(uv_rwlock_t* rwlock);`

其中 `uv_rwlock_rdlock` 和 `uv_rwlock_rdunlock` 为读锁的加解锁, `uv_rwlock_wrlock` 和 `uv_rwlock_wrunlock` 是写锁的加解锁。

读者也能通过改造睡眠排序的错误代码来初步了解一下 libuv 的读 / 写锁。读者可以打开随书代码“32. libuv sleep sort advanced”中的 `rwlock/sleep.cc` 阅读, 或者跟着下文的步骤来处理先前的错误代码。

首先依旧是 `ThreadArg` 结构体的改造。

```
struct ThreadArg {
    ...
    uv_rwlock_t* rwlock_handle;
};
```

然后是 `Sort` 函数中的改造。

```
...

std::vector<uint32_t> vec;
std::vector<uv_thread_t> handles(array->Length());
uv_rwlock_t handle;
uv_rwlock_init(&handle);
for(uint32_t i = 0; i < array->Length(); i++)
{
    ThreadArg* arg = new ThreadArg();
    ...
    arg->rwlock_handle = &handle;
    uv_thread_create(&handles[i], Sleep, arg);
}

...
uv_rwlock_destroy(&handle);
```

最后, 依旧是改造线程函数。

```

void Sleep(void* _)
{
    ThreadArg* arg = (ThreadArg*)_;

    // 睡上 num * 100 毫秒
    _Sleep(arg->num * 100);

    uv_rwlock_wrlock(arg->rwlock_handle);
    arg->vec->push_back(arg->num);
    uv_rwlock_wrunlock(arg->rwlock_handle);

    // 测试读锁
    uv_rwlock_rdlock(arg->rwlock_handle);
    printf("当前数组长度: %lu\n", arg->vec->size());
    uv_rwlock_rdunlock(arg->rwlock_handle);

    delete arg;
}

```

为了演示效果更佳，笔者在该函数的后面加上了读锁的测试。接下来运行几遍看看吧。

```

$ node-gyp rebuild
...
$ node
> const addon = require("./build/Release/mutex_sleep");
undefined
> addon.sort([3,2,1,3]);
当前数组长度: 1
当前数组长度: 2
当前数组长度: 4
当前数组长度: 4
[ 1, 2, 3, 3 ] # 阻塞 300 毫秒左右

```

这里的结果笔者不解释了。下面讲一下读锁的相关输出。当读者多运行几次后，会发现测试输出的数组长度有时候依次为 1、2、3、4，有时候又是 1、2、4、4。因为当线程从写锁中解放出来时，有两种可能：一种可能是另一个线程抢占，写锁马上被锁住了，这个时候先前线程的读锁就又是等待状态了，等到另一个线程的写锁解锁后也就是 4 个元素都被推入了数组容器，剩下两个读锁中的流程读出来的数组长度自然就都是 4 了；另一种可能就是写锁解锁后，同线程的读锁马上被锁住，另一个线程处于等待状态，这时数组容器的长度就是 3，当执行完后另一个线程继续执行写锁和读锁的内容，输出的长度自然就是 4 了。

3. 信号量 (Semaphore)

为了防止出现因多个程序同时访问一个共享资源而引发的一系列问题,我们需要一种方法,它可以通过生成并使用令牌来授权,在任一时刻只能有一个执行线程访问代码的临界区域。临界区域是指执行数据更新的代码需要独占式地执行。而信号量就可以提供这样的一种访问机制,让一个临界区同一时间只有一个线程在访问它,也就是说信号量是用来调协线程对共享资源的访问的。

信号量(Semaphore)又被称为旗语,其是一个同步原语,程序对其进行的访问都是原子操作,且只允许对它进行等待(即 P(信号变量))和发送(即 V(信号变量))信息操作。最简单的信号量是只能取 0 和 1 的变量,这也是信号量最常见的一种形式,叫作二进制信号量。而可以取多个正整数的信号量被称为通用信号量。

libuv 中的信号量相关函数与 pthread 中的相关函数类似。其句柄类型为 uv_sem_t, 相关函数如下:

- int uv_sem_init(uv_sem_t* sem, unsigned int value);
- void uv_sem_destroy(uv_sem_t* sem);
- void uv_sem_post(uv_sem_t* sem);
- void uv_sem_wait(uv_sem_t* sem);
- int uv_sem_trywait(uv_sem_t* sem)。

其中初始化函数中的参数 value 表示信号量的计数值。

uv_sem_post() 函数为发送信号, uv_sem_wait() 为等待信号。我们改造一下刚才的错误代码,保存至“32. libuv sleep sort advanced”的 sem/sleep.cc 中。

首先依旧是 ThreadArg 的改造。

```
struct ThreadArg {
    ...
    uv_sem_t* sem_handle;
};
```

然后是 Sort 函数。

```
std::vector<uint32_t> vec;
std::vector<uv_thread_t> handles(array->Length());
uv_sem_t handle;
uv_sem_init(&handle, 1); // 初始化计数为 1, 表示最多只有一个临界区能同时访问
for(uint32_t i = 0; i < array->Length(); i++)
```

```
{
    ThreadArg* arg = new ThreadArg();
    ...
    arg->sem_handle = &handle;
    uv_thread_create(&handles[i], Sleep, arg);
}

...
uv_sem_destroy(&handle);
```

最后依旧是线程函数。

```
void Sleep(void* _)
{
    ThreadArg* arg = (ThreadArg*)_;

    // 睡上 num * 100 毫秒
    _Sleep(arg->num * 100);

    uv_sem_wait(arg->sem_handle); // 等待信号量并进入临界区使得计数减一
    arg->vec->push_back(arg->num);
    uv_sem_post(arg->sem_handle); // 发送信号量使得计数加一，以供其他临界区进入

    delete arg;
}
```

下面还是试试运行这段代码吧。

```
$ node-gyp rebuild
...
$ node
> const addon = require("./build/Release/sem_sleep");
undefined
> addon.sort([3,2,1,3]);
[ 1, 2, 3, 3 ] # 阻塞 300 毫秒左右
```

4. 条件变量（Condition Variable）与屏障（Barrier）

libuv 中的条件变量和屏障可以直接对应于 pthread 中的相应内容，在此就不详细展开讲解了。

有兴趣的读者可以阅读 libuv 官方 API 文档获取更多的信息，也可以阅读《POSIX 多线程程序设计》一书以了解这些多线程编程的前置知识体系。

6.2.3 工作队列

在 libuv 中，工作队列指的是 `uv_queue_work()` 这一系列的函数。它可以使得你的程序使用额外的线程来执行异步任务，并在任务完成后触发回调函数。

阅读过 5.5.1 节的读者可能会有印象，NAN 中的 `AsyncQueueWork` 做的就是与此类似的事情。实际上，NAN 的这个 `AsyncQueueWorker` 用的就是 `uv_queue_work()`。

```
// AsyncQueueWorker 函数实现
inline void AsyncQueueWorker (AsyncWorker* worker) {
    uv_queue_work(
        uv_default_loop()
        , &worker->request
        , AsyncExecute
        , reinterpret_cast<uv_after_work_cb>(AsyncExecuteComplete)
    );
}
```

笔者接下来对应着 `uv_queue_worker()` 函数声明讲解一下吧。

```
int uv_queue_work(uv_loop_t*, uv_work_t*, uv_work_cb, uv_after_work_cb);
```

- `uv_loop_t*`: libuv 的事件循环句柄。
- `uv_work_t*`: 工作句柄，其中存储了一些必要的的数据，用于传入工作线程执行。
- `uv_work_cb`: 用于在额外线程中执行的函数，原型是 `void (*uv_work_cb)(uv_work_t* req)`。
- `uv_after_work_cb`: 工作线程执行完之后的回调函数。

我们就着上面的 `AsyncQueueWorker` 和其他的几个函数一起来看看它们是怎么串起来的。

```
inline void AsyncExecute (uv_work_t* req) {
    AsyncWorker *worker = static_cast<AsyncWorker*>(req->data);
    worker->Execute();
}

inline void AsyncExecuteComplete (uv_work_t* req) {
    AsyncWorker* worker = static_cast<AsyncWorker*>(req->data);
    worker->WorkComplete();
    worker->Destroy();
}
```

如上述代码中所示，在 `AsyncQueueWorker` 中传入的 `uv_work_cb` 就是 `AsyncExecute`，在额外的线程中，这个函数将 `uv_work_t* req` 中的 `data` 转换回笔者事先定义好的 `AsyncWorker` 对象实例中，然后执行它的 `Execute()` 函数。等到这个函数执行完了，就说明工作线程的工作完成了，那么会执行传入的 `uv_after_work_cb` 也就是 `AsyncExecuteComplete` 函数，在回调中执行 `worker->WorkComplete()` 和 `Destroy()` 两个函数。前面讲过，`WorkComplete()` 函数是一个虚函数，需要大家在子类中继承，通常它做的事情就是执行从 JavaScript 中传过来的回调函数。

不过到此为止，有些人可能还会纠结于为什么 `req->data` 转过来就是 `AsyncWorker`，其实这在 `AsyncWorker` 的构造函数中有体现。

```
// AsyncWorker 构造函数节选
// 其中 request 是它的成员变量，一个 `uv_work_t` 句柄
AsyncWorker::AsyncWorker(Callback *callback_)
    : callback(callback_), errmsg_(NULL) {
    request.data = this;

    ...
}
```

由于本节内容基本上可以通过 NAN 来达到同样的效果，因此样例代码就不放出了。感兴趣的读者可以自行参考 NAN 中的 `AsyncWorker` 代码研究一下。

6.2.4 小结

本节介绍了 libuv 中一些跨线程编程的基础概念，如 libuv 线程的原理以及同步原语。

其中同步原语如下：

- 互斥锁；
- 读 / 写锁；
- 信号量；
- 条件变量；
- 屏障。

对于条件变量和屏障这两个同步原语，本书并未做过多解释，读者可以阅读其他文献了解详细信息。

除了这些基础知识之外，本节还介绍了 NAN 中 `AsyncWorker` 的原理。其就是调用了 libuv 中的工作队列 `uv_queue_work()` 来完成操作的。

6.2.5 参考资料

- [1] An Introduction to libuv - THREADS: <http://nikhilm.github.io/uvbook/threads.html>.
- [2] Genius sorting algorithm: Sleep sort: https://bl0ckeduser.github.io/sleepsort/sleep_sort_trimmed.html.
- [3] C++ Access to vector from multiple threads: <https://stackoverflow.com/questions/12260946/c-access-to-vector-from-multiple-threads?answertab=active#answer-12261077>.
- [4] Linux 内核同步原语之原子操作: <http://blog.csdn.net/npylp/article/details/7262388>.
- [5] 线程安全——Scope Lock 模式: <https://xcoder.in/2012/09/08/scope-lock/>.
- [6] Linux 进程间通信——使用信号量: <http://blog.csdn.net/ljianhui/article/details/10243617>.
- [7] 信号量和互斥量 C 语言示例理解线程同步: <http://www.cnblogs.com/nisen/p/6083866.html>.
- [8] (美) 布滕霍夫. POSIX 多线程程序设计 [M]. 北京: 中国电力出版社, 2003.
- [9] uv_async_t — Async Handle — libuv documentation: <http://docs.libuv.org/en/v1.x/async.html>.

6.3 跨线程通信

本节内容中说的跨线程通信，可不只是 libuv 自维护的一堆线程间的通信，实际上可以是开发者任意统一进程间各线程的通信。

下面举个简单的例子，在阿里云 ONS 的 C++ SDK 中，会自行启动线程监听从远端而来的消息，开发者为消息的到来设置一个回调函数，只不过这些回调函数都是执行在 ONS SDK 自己的线程中的。我们如果需要将它集成到 Node.js 中，就需要在其线程中与 Node.js 主事件循环的线程进行通信。

6.3.1 uv_async_t 句柄

而在 libuv 中，做这件事情就需要靠 uv_async_t 这个句柄了。

Async 句柄允许用户从别的线程中“唤起”事件循环，并在事件循环中执行一个它指定的回调函数。¹

¹ <http://docs.libuv.org/en/v1.x/async.html#uv-async-t-async-handle>

关于 Async 句柄，有两个主要的函数。

- `int uv_async_init(uv_loop_t*, uv_async_t*, uv_async_cb)`: 初始化一个 Async 句柄，最后的 `uv_async_cb` 可以是一个空指针即 `NULL`，表示每次通过 `uv_async_send` 唤起事件循环的时候要执行的回调函数。
- `int uv_async_send(uv_async_t*)`: 唤起事件循环，并在事件循环中执行事先通过 `uv_async_init` 函数设定好的回调函数。

注意：libuv 会合并对 `uv_async_send()` 的调用，也就是说并不是每次调用它都会产生回调的执行。例如在调用回调之前 `uv_async_send()` 被连续调用 5 次，那么实际上回调只会被调用一次。但是如果在回调执行之后再执行 `uv_async_send()`，那么回调将会再次被调用。¹

为了使得大家更有共鸣，笔者对于 `uv_async_t` 使用一个样例进行解析，并且该样例的场景也正好是前文中说过的一个场景——假设我们有一个非常方便的 C++ 库，但是它有一套线程机制，它的结果回调是在另一个线程中被触发的，我们将要把库中线程的结果插回 Node.js 主事件循环中。

不过由于篇幅的原因，该样例在本节中将会是一个半成品——只是用于向大家说明 `uv_async_t` 句柄的用法。

6.3.2 Watchdog 半成品实战解析

本节的随书代码在“33. libuv watchdog”中。

首先该样例做的事情是用于监听一个目录下的文件变化，其有些类似于 `fsevents`² 这个包，在完成于 Node.js 中调用时将会是类似这样的：

```
watchdog.watch('/tmp', function(watchId, dir, filename, action, oldFilename)
{
    // ...
});
```

当 `/tmp` 目录有变化（如文件新建、修改、移动等）时，会触发后面的回调函数。

¹ http://docs.libuv.org/en/v1.x/async.html#c.uv_async_send

² <https://www.npmjs.com/package/fsevents>

1. 文件系统监听库——efsw

在 C++ 中，做这么一个监听的活儿有各种各样的方法，本书中为了讲解方便，用了一个符合笔者所说场景的开源库 efsw¹。

efsw 是一个 C++ 的跨平台文件系统监视和通知库，由 Martín Lucas Golini² 开发维护，它会异步地监视文件系统中指定目录下的文件或者目录的修改，并在获得这些信息的同时触发一个事件。efsw 支持递归的目录监视，跟踪整个子目录树。其当前支持如下：

- Linux：通过 inotify³。
- Windows：通过 I/O 完成端口。
- macOS：通过 FSEvents⁴ 或者 kqueue。
- FreeBSD / BSD：通过 kqueue。

efsw 最初是 simplefilewatcher⁵ 的一个 Fork 分支。

efsw 的基础用法非常简单，首先基于 efsw::FileWatchListener 继承一个监听类，其中需要实现 handleFileAction 函数，这个函数在 efsw 有事件被触发的时候在其线程中被调用，参数如下。

- efsw::WatchID watchid：监听编号，WatchID 实际上是一个 long 类型的 typedef。
- const std::string& dir：监听的目录名。
- const std::string& filename：监听目录下这次事件相关的文件名（如在该目录下新建 1.txt，那么它的值就是 "1.txt"）。
- efsw::Action action：事件类型，是一个枚举。
 - efsw::Action::Add：值为 1，创建事件。
 - efsw::Action::Delete：值为 2，删除事件。
 - efsw::Action::Modified：值为 3，修改事件。
 - efsw::Action::Moved：值为 4，移动事件。
- std::string oldFilename：事件相关文件名的原文件名（适用于文件移动事件等）。

1 项目地址为 <https://bitbucket.org/SpartanJ/efsw>。本书中随书代码用的是第三方在 GitHub 上的一个镜像 <https://github.com/aneutron/efsw>，代码不一定是最新的，但大致能满足我们的需求。

2 阿根廷研发工程师，Ensoft 联合创始人。

3 inotify 是一个 Linux 特性，它监控文件系统操作，比如读取、写入和创建。

4 https://developer.apple.com/library/content/documentation/Darwin/Conceptual/FSEvents_ProgGuide/UsingtheFSEventsFramework/UsingtheFSEventsFramework.html

5 由 James Wynn 开发，项目地址是 <https://code.google.com/archive/p/simplefilewatcher/>。

这是在 efsw 的 README.md 中给出的一个样例子类。

```
class UpdateListener : public efsw::FileWatchListener
{
public:
    UpdateListener() {}

    void handleFileAction( efsw::WatchID watchid, const std::string& dir,
        const std::string& filename, efsw::Action action, std::string oldFilename =
        "" )
    {
        switch( action )
        {
            case efsw::Actions::Add:
                std::cout << "DIR (" << dir << ") FILE (" << filename << ") has
event Added" << std::endl;
                break;
            case efsw::Actions::Delete:
                std::cout << "DIR (" << dir << ") FILE (" << filename << ") has
event Delete" << std::endl;
                break;
            case efsw::Actions::Modified:
                std::cout << "DIR (" << dir << ") FILE (" << filename << ") has
event Modified" << std::endl;
                break;
            case efsw::Actions::Moved:
                std::cout << "DIR (" << dir << ") FILE (" << filename << ")
has event Moved from (" << oldFilename << ")" << std::endl;
                break;
            default:
                std::cout << "Should never happen!" << std::endl;
        }
    }
};
```

在定义好事件监听类之后，只需要实例化一个监听器，然后不断往里面添加目录监听并指定该目录监听所对应的监听类即可，如：

```
efsw::FileWatcher * fileWatcher = new efsw::FileWatcher();
UpdateListener * listener = new UpdateListener();

// 第三个参数为 `true`，代表子目录递归监听
efsw::WatchID watchID = fileWatcher->addWatch( "/tmp", listener, true );

// 这是一次非递归监听
```



```
efsw::WatchID watchID2 = fileWatcher->addWatch( "/usr", listener, false );

// 开始异步监听
fileWatcher.watch();

// 移除对 `/usr` 的监听
fileWatcher->removeWatch( watchID2 );
```

关于 efsw 的更多用法大家可以去往其项目主页了解更多信息，也可以阅读它的源码进行解读。

2. 相应的 binding.gyp

既然用了第三方库，那么很常见的一种做法是先把这个库编译成一个静态链接库，就如 3.6.3 节对应的随书代码“6. mapped property interceptor”等一样。

大家可以打开“33. libuv watchdog”中的 binding.gyp 看看 targets:

```
{
  "targets": [{
    "target_name": "efsw",
    "type": "static_library",
    "sources": [
      "../deps/efsw/src/efsw/Debug.cpp",
      "../deps/efsw/src/efsw/DirectorySnapshot.cpp",
      "../deps/efsw/src/efsw/DirectorySnapshotDiff.cpp",
      "../deps/efsw/src/efsw/DirWatcherGeneric.cpp",
      "../deps/efsw/src/efsw/FileInfo.cpp",
      ...
      "../deps/efsw/src/efsw/Thread.cpp",
      "../deps/efsw/src/efsw/Watcher.cpp",
      "../deps/efsw/src/efsw/WatcherFSEvents.cpp",
      "../deps/efsw/src/efsw/WatcherGeneric.cpp",
      "../deps/efsw/src/efsw/WatcherInotify.cpp"
    ],
    "include_dirs": [
      "../deps/efsw/include",
      "../deps/efsw/src"
    ],
    ...
  }]
}
```

这里有几点注意事项:

- 目标名为 "efsw";
- 目标类型为静态链接库;
- 指定了源文件路径;
- 指定了包含目录路径。

再往下看是一个条件块, 分别指明了 Windows 下以及非 Windows 系统下各需要包含的文件。还有就是 macOS 系统下需要额外指定给链接器的 FSEvents 编译选项。

```
"conditions": [
  ["OS=="win\``, {
    "sources": [
      "../deps/efsw/src/efsw/platform/win/FileSystemImpl.cpp",
      "../deps/efsw/src/efsw/platform/win/MutexImpl.cpp",
      "../deps/efsw/src/efsw/platform/win/SystemImpl.cpp",
      "../deps/efsw/src/efsw/platform/win/ThreadImpl.cpp",
      "../deps/efsw/src/efsw/FileWatcherWin32.cpp",
      "../deps/efsw/src/efsw/WatcherWin32.cpp"
    ]
  }],
  ["OS!="win\``, {
    "sources": [
      "../deps/efsw/src/efsw/platform/posix/FileSystemImpl.cpp",
      "../deps/efsw/src/efsw/platform/posix/MutexImpl.cpp",
      "../deps/efsw/src/efsw/platform/posix/SystemImpl.cpp",
      "../deps/efsw/src/efsw/platform/posix/ThreadImpl.cpp",
      "../deps/efsw/src/efsw/FileWatcherKQueue.cpp",
      "../deps/efsw/src/efsw/WatcherKQueue.cpp"
    ]
  }],
  ["OS=="mac\``, {
    "defines": [
      "EFSW_FSEVENTS_SUPPORTED"
    ],
    "xcode_settings": {
      "OTHER_LDFLAGS": [
        "-framework CoreFoundation -framework CoreServices"
      ]
    }
  }
]
}]
]
```

efsw 静态链接库的 GYP 配置写好之后, 就是主体 Watchdog 了。

```
{
  "target_name": "watchdog",
  "sources": [
    "watchdog.cc"
  ],
  "include_dirs": [
    "<!(node -e \"require('nan')\")\" "
  ],
  "dependencies": [
    "efsw"
  ],
  "conditions": [
    ["OS==\"mac\"", {
      "xcode_settings": {
        "OTHER_LDFLAGS": [
          "-framework CoreFoundation -framework CoreServices"
        ]
      }
    }]
  ]
}
```

其非常简洁明了。但是有一点需要注意的是，在这个 Node.js 扩展模块中，macOS 系统下编译一样要加上 FSEvents 的支持。

3. Watchdog 代码解读

在看完了 binding.gyp 之后，我们可以开始看 Watchdog 的源码了。

需要注意的是，本节中的代码是一个半成品，很多地方没有做细节上的优化。比如没有取消监听的函数；比如 FileWatcher 对象一旦从堆内存生成，再也不释放它；等等。

明白了上面的注意事项，就可以打开随书代码“33. libuv watchdog”中的 watchdog.cc 了。

为了代码风格的统一性，在代码中笔者通过宏定义把一些小驼峰的函数名转换成了大驼峰形式。

首先在模块导出函数中，笔者初始化了全局的 file_watcher 对象，并开始监听。

```
efsw::FileWatcher* file_watcher;

...

NAN_MODULE_INIT(Init)
{
```

```
// 半成品, 没有地方回收该指针
file_watcher = new efsw::FileWatcher();
file_watcher->Watch();

Nan::Export(target, "watch", Watch);
}
```

然后将目光转向 `Watch` 函数。这个函数在 JavaScript 中使用的方式如前文所示, 第一个参数是监听路径, 第二个参数是事件回调函数。笔者在函数中判断了参数合法性之后, 将路径 `path` 抽取出来以提供 `std::string` 的路径, 然后将原回调函数生成一个 `Nan::Callback` 类型的回调函数。

在这些步骤做好之后, 声明一个 `UpdateListener` 类的对象, 并把回调函数传入, 然后把这个对象加入需要管理的一个 `std::map` 中 (不过在半成品中并没有回收和管理这些对象的逻辑)。

接下来就是通过 `file_watcher` 的 `AddWatch` 函数将该路径加入监听行列中, 传入路径 `*path` 和监听器 `listener`。这个函数返回大于 0 的值表示监听成功并且返回的是一个 `watch_id`。若小于 0 则说明出错了, 笔者会通过 `efsw::Errors::Log::GetLastErrorLog()` 获取错误, 然后抛到 Node.js 层。

最后, 返回给 Node.js 一个 `watch_id` 即可。

```
NAN_METHOD(Watch)
{
    if(info.Length() < 2 || !info[0]->IsString() || !info[1]->IsFunction())
    {
        return Nan::ThrowError("Wrong argument");
    }

    Nan::Utf8String path(info[0]->ToString());
    Nan::Callback* callback = new Nan::Callback(info[1].As<v8::Function>());

    // 半成品, 没有地方回收该指针
    UpdateListener* listener = new UpdateListener(callback);
    listeners[*path].push_back(listener);

    efsw::WatchID watch_id = file_watcher->AddWatch(*path, listener, true);

    if(watch_id < 0)
    {
        return Nan::ThrowError(efsw::Errors::Log::GetLastErrorLog().c_str());
    }
}
```

```

    }

    info.GetReturnValue().Set(Nan::New((int)watch_id));
}

```

讲完了 Watch 函数，我们再来看看监听器类所实现的内容。在本节中监听器的 HandleFileAction 函数的作用就是创建一个 uv_async_t 句柄，然后将该次事件中的各种参数（如目录名、事件类型等）集合起来，通过 uv_async_send() 函数将这些内容随着 Node.js 主事件循环的唤起一起发到 uv_async_t 句柄的回调函数中处理。

在讲 UpdateListener 前，有必要给大家展示一下我们将随着 uv_async_send() 函数一起发过去的数据的类型结构。

```

struct AsyncArgs {
    efsw::WatchID watch_id;
    std::string dir;
    std::string filename;
    efsw::Action action;
    std::string old_filename;
    uv_async_t handle;
    Nan::Callback* callback;
};

```

这个 AsyncArgs 实例化成对象之后，指针会被强制转换成 void* 类型并赋值得到 uv_async_t 句柄的 data 成员变量中。我们接下来看看监听器吧。

```

class UpdateListener : public efsw::FileWatchListener {
public:
    UpdateListener(Nan::Callback* callback) :
        callback(callback)
    {
    }

    ~UpdateListener()
    {
        Nan::HandleScope scope;
        delete callback;
    }

    void HandleFileAction(
        efsw::WatchID watchid,
        const std::string& dir,
        const std::string& filename,

```

```

        efsw::Action action,
        std::string old_filename)
    {
        AsyncArgs* args = new AsyncArgs();
        args->watch_id = watchid;
        args->dir = dir;
        args->filename = filename;
        args->action = action;
        args->old_filename = old_filename;
        args->callback = callback;
        uv_async_init(uv_default_loop(), &args->handle, OnEvent);
        args->handle.data = (void*)args;
        uv_async_send(&args->handle);
    }

private:
    Nan::Callback* callback;
};

```

如大家所见，UpdateListener 继承自 efsw::FileWatchListener，在构造函数中传入与这次监听相关的事件回调函数 callback。在析构函数中将这个回调函数删除。然后在 HandleFileAction 函数中，笔者新建了一个 AsyncArgs* 对象，将连同 watch_id 等各相关参数在内的内容都赋值给该对象，再通过 uv_async_init() 函数初始化 AsyncArgs* 中的 uv_async_t 句柄，绑上 OnEvent 函数。做好了这些事情之后，笔者把这个 AsyncArgs* 对象赋值给 uv_async_t 句柄的 data 成员变量，最后通过 uv_async_send() 唤起主事件循环，进入 OnEvent 函数中。

注意：在别的非主事件循环的线程中（如 efsw 的事件触发所在的线程）调用 Node.js 中的 V8 对象都是不安全的，所以不推荐在其他线程做任何 V8 相关的操作。在本节的例子中也一样，HandleFileAction 只做底层的数据传输，并没有事先把各种数据转换为 V8 的数据。

```

void OnEvent(uv_async_t* handle)
{
    Nan::HandleScope scope;
    AsyncArgs* data = (AsyncArgs*)handle->data;
    Local<Value> argv[] = {
        Nan::New((int)data->watch_id),
        Nan::New(data->dir.c_str()).ToLocalChecked(),
        Nan::New(data->filename.c_str()).ToLocalChecked(),
        Nan::New(data->action),
        Nan::New(data->old_filename.c_str()).ToLocalChecked()
    };
};

```

```

data->callback->Call(5, argv);
uv_close((uv_handle_t*)handle, OnAsyncClosed);
}

```

OnEvent() 函数中的逻辑就比较简单、粗暴了。在句柄作用域“光芒”的笼罩下，把 uv_async_t 句柄的 data 转换回 AsyncArgs*，然后把其中所需要的成员变量转换成 V8 的数据类型，之后通过调用 data->callback->Call() 发起回调函数的调用。这样，开发者就在自己 JavaScript 层的代码中开始执行回调函数的内容了。在做完这一系列的事情之后，记得将句柄关闭并且销毁，即将待关闭句柄（也就是 uv_async_t）以及句柄关闭完成所对应的回调函数传入 uv_close() 函数进行调用。在句柄被关闭后，OnAsyncClosed 回调函数就会被执行。

```

void OnAsyncClosed(uv_handle_t* handle)
{
    uv_async_t* async = (uv_async_t*)handle;
    AsyncArgs* data = (AsyncArgs*)async->data;
    delete data;
}

```

在句柄关闭回调函数中，笔者做的就是将 AsyncArgs* 通过 delete 进行回收。

6.3.3 Watchdog 试运行

在解析完 Watchdog 的代码之后，我们可以运行一下这个半成品来看看效果了。

```

$ node-gyp rebuild
...
$ node
> const watchdog = require("./build/Release/watchdog")
undefined
> watchdog.watch("/tmp", function() { console.log(arguments); });
1
> watchdog.watch("/tmp/sdfkjl", function() { console.log(arguments); });
Error: File not found ( /tmp/asfd )
    at Error (native)
    at repl:1:10
    at sigintHandlersWrap (vm.js:96:12)
    at REPLServer.defaultEval (repl.js:313:29)
    at bound (domain.js:280:14)
    at REPLServer.<anonymous> (repl.js:513:10)
    at emitOne (events.js:101:20)
    at REPLServer.emit (events.js:188:7)
    at REPLServer.Interface._onLine (readline.js:239:10)
    at REPLServer.Interface._line (readline.js:585:8)

```

在我们第一次执行 `watchdog.watch()` 的时候，函数返回了一个整数 1，这就是这次监听的 `WatchId` 了。而第二次我们再次监听的时候，Node.js 就抛出了一个错误：`File not found (/tmp/asfd)`。这个错误是因为监听目录不存在，由 `file_watcher->AddWatch()` 返回的，然后我们通过 `efsw::Errors::Log::GetLastErrorLog().c_str()` 拿到错误的文字描述后用 `Nan::ThrowError` 抛出来的。

在代码执行完之后，保持这个姿势不要动，我们尝试到 `/tmp` 目录下执行一些操作（如果是 Windows 用户，则可以尝试其他合法路径）。

- 新建一个文件，如 `test`：

```
{ '0': 1, '1': '/tmp/', '2': 'test', '3': 1, '4': '' }
```

- 修改刚才 `test` 的内容：

```
{ '0': 1, '1': '/tmp/', '2': 'test', '3': 3, '4': '' }
```

- 移动刚才的 `test` 到 `test.bak`：

```
{ '0': 1,
  '1': '/tmp/',
  '2': 'test.bak',
  '3': 4,
  '4': 'test' }
```

- 删除 `test.bak`：

```
{ '0': 1, '1': '/private/tmp/', '2': 'test.bak', '3': 2, '4': '' }
```

其结果应该在大家的预料之内（除了有时候由于系统内部的操作，比如在创建文件的时候，可能还会带上一个修改的事件）。

至此，Watchdog 的半成品就讲解完毕了。相信大家经过本节内容之后，对 `uv_async_t` 句柄有了一定程度的了解。

6.3.4 小结

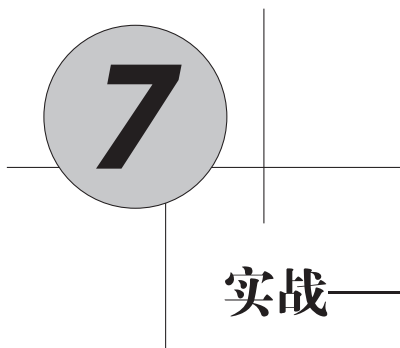
本节主要讲解了在 libuv 中，其他线程如何与 libuv 的事件循环进行通信。落地到 Node.js 中，其实就相当于如何在一个非主事件循环的线程中将信息传达到 Node.js 的 JavaScript 层（即主事件循环）代码。在这种场景中我们就需要用到 `uv_async_t` 这个句柄了。

之后辅以一个半成品 Watchdog 文件监视模块来对 `uv_async_t` 进行一次实战，这使得大家心中不再觉得其很抽象。

感谢开源贡献者们为本节提供了一个非常酷的库——`cfsw`。

6.3.5 参考资料

[1] objective c - fsevents command line - Stack Overflow: <https://stackoverflow.com/questions/15752583/fsevents-command-line/15752705#15752705>.



实战——文件监视器

在第 6 章的最后，笔者实现了一个半成品的文件监视器，用于说明 `uv_async_t` 是怎么在项目中的使用的。在本章中笔者将实现一个完整的文件监视器包，让大家从中获得成就感。

在本章中笔者会尽量脱离第 6 章的一些讲解，但是有些复用的地方还是推荐读者使用第 6 章的内容，大家可回过头去阅读。

7.1 准备工作

“工欲善其事，必先利其器”，下面为我们所需要进行的工作做一些准备和调研，这是非常有必要的。

7.1.1 功能规划

我们要做的是个文件监视器，监视一个目录里面的文件或者目录的变更事件，如新建、删除以及修改等。我们有一个标杆，就是 macOS 下面的一个 Node.js 包，叫 `fsevents`。

读者可以参考 <https://github.com/XadillaX/node-efsw> 或本书随书代码的“34. efsw”（实际上是 `node-efsw` 这个包的一个 Git 节点），或者自行创建一个目录，跟着本书按部就班来实现。

1. 参考 fsevents

fsevents 是 macOS 系统下用于获取 FSEvents¹ 的一个 Node.js 原生包。其中 macOS 下的 FSEvents API 允许应用注册一个监听到一个目录树上，当目录树有修改时就会得到一个消息。

这个包的地址是 <https://www.npmjs.com/package/fsevents>，在它的文档中有介绍用法。其中一个简单的用法是这样的：

```
var fsevents = require('fsevents');
var watcher = fsevents(__dirname);
watcher.on('fsevent', function(path, flags, id) { }); // 由 macOS 发出的原始事件
watcher.on('change', function(path, info) { }); // 所有修改的通用事件
watcher.start() // 开始监听
watcher.stop()  // 停止监听
```

2. API 假想

我们的目标则是实现一个类似于 fsevents 的但是功能并不需要那么多的包，并且跨平台。

假设我们的包叫 efsw，那么在使用的時候可能是这样的：

```
const Watcher = require('efsw').Watcher;
const watcher = new Watcher(__dirname);
watcher.on('change', function(path, info) {
  console.log(path, info.action, info.old);
});
```

综上所述，我们的 Watcher 是一个类，并且继承自 EventEmitter。一旦有文件变动则会触发一个 change 事件，传入相应的文件名，以及一些基本信息。

3. 代码分工

既然是一套真的能跑而且相对健壮的包，那么代码自然需要有各种错误处理等内容。而在很多情况下，使用纯 C++ 来开发不太方便，而且在 C++ 下继承 EventEmitter 也比较麻烦，所以代码自然就有了 Node.js 层面和 C++ 层面的分工。

在本书中，笔者是这么分工的，如图 7-1 所示。

¹ https://developer.apple.com/library/content/documentation/Darwin/Conceptual/FSEvents_ProgGuide/UsingtheFSEventsFramework/UsingtheFSEventsFramework.html

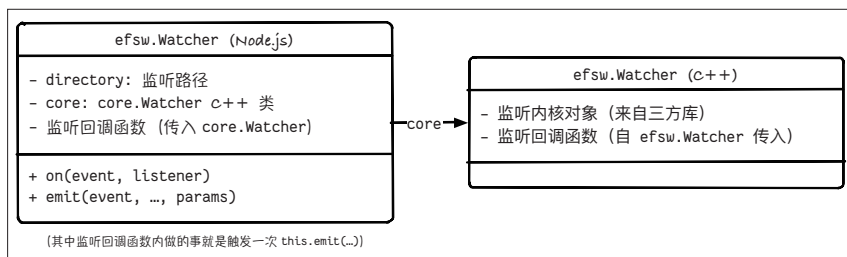


图 7-1 efsw 代码层面的分工

- `efsw.Watcher`: 这应该是一个 JavaScript 层面的类，继承自 `EventEmitter`，所以它有 `on` 和 `emit` 等函数。其次，除了有 `directory` 等基本属性之外，它还包含了以下两个成员属性。
- `core`: 暂定名，它是一个 C++ 模块的内核，里面是真的用于监听文件系统变化的逻辑，如 macOS 下就调用 `FSEvents` 系列的 API，这将会是一个 C++ 下的 `Node.js` 类，这种类的实现 5.4.4 节也曾提到过。
- 监听回调函数：该函数是用于传给 `core` 属性的。当 `core` 的逻辑中监听到了文件系统的变化时，就会触发这个回调函数。而这个回调函数中暂定的逻辑就是实现一些基础的计算，把原始的事件数据加工成 `Node.js` 中使用的比较友好的数据，并通过 `.emit()` 来触发一下 `change` 事件。
- `core.Watcher`: 这将会是一个使用 C++ 实现的类，基于 `Nan::ObjectWrap` 实现，其中包含了监听的核心逻辑——该逻辑将来自一个第三方库（阅读了 6.3.2 节的读者应该知道，笔者将使用 `efsw`¹ 这个库）。

4. 包的初始化

一个 `Node.js` 的包初始化会借助于 `NPM` 的 `CLI` 命令，但也可以自行在项目的目录下创建一个 `package.json` 来搞定。

```

$ npm init
...

name: (efsw)
version: (1.0.0)
description: Node.js binding for efsw.
entry point: (index.js) efsw.js
test command:
git repository: (https://github.com/XadillaX/node-efsw.git)

```

¹ <https://bitbucket.org/SpartanJ/efsw>。

```

keywords: efs, watcher, fsevents
author: XadillaX <i@2333.moe>
license: (ISC) MIT
About to write to /Path/efs/package.json:

{
  "name": "efs",
  "version": "1.0.0",
  "description": "Node.js binding for efs.",
  "main": "efs.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/XadillaX/node-efs.git"
  },
  "keywords": [
    "efs",
    "watcher",
    "fsevents"
  ],
  "author": "XadillaX <i@2333.moe>",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/XadillaX/node-efs/issues"
  },
  "homepage": "https://github.com/XadillaX/node-efs#readme"
}

Is this ok? (yes)

```

这时目录下就有了一个生成好的 `package.json`，大家可以自行修改里面的一些字段。

7.1.2 文件系统监听库——efs

efs 是一个 C++ 的跨平台文件系统监视和通知库。关于 efs 这个库，大家可以阅读 6.3.2 节获取详细的信息。

1. 克隆 efsw 到本地

我们在项目目录下新建一个目录用于存放第三方库的源码，笔者在此使用的是 `src/deps/`，然后在该项目目录下，把 efsw 的源码¹克隆到本地。

```
$ mkdir src && mkdir src/deps # 如果你的目录还没有创建就执行这条命令
$ cd src/deps # 进入 src/deps 目录
$ git clone https://github.com/XadillaX/efsw.git
```

等到 efsw 这个第三方库克隆到本地之后，目录结构大致如下：

```
.
├── LICENSE           // 这个文件可以忽略
├── README.md         // 这个文件可以忽略
├── package.json
└── src
    ├── deps
    └── efsw
```

如果大家新建的项目目录是一个 Git 下的目录，则可以在项目的根目录下新建一个 `.gitmodules` 的文件来声明这个第三方的 Git 仓库目录。

```
; .gitmodules 文件

[submodule "src/deps/efsw"]
    path = src/deps/efsw
    url = https://github.com/XadillaX/efsw
```

2. 关于 binding.gyp 的创建

既然是一个第三方库，那笔者当然希望将其以静态链接库的形式事先编译好，作为我们所要写的 C++ 模块的依赖存在。这样笔者就要为其单独写一块声明，详情可以参考 6.3.2 节，只不过需要将其 `binding.gyp` 中的 efsw 源文件路径修正成当前项目所对应的相对路径，例如：

```
"../deps/efsw/src/efsw/Thread.cpp"
```

应为

```
"./src/deps/efsw/src/efsw/Thread.cpp"
```

¹ 由于 efsw 的原始源码以 hg 的形式维护在 BitBucket 下，因此笔者在 GitHub 上建立了一个它的镜像，使得大家能方便地使用 Git 进行操作，其仓库地址是 <https://github.com/XadillaX/efsw.git>。

那么，关于 efsw 的 binding.gyp 声明结果应该是类似于这样的：

```
{
  "targets": [{
    "target_name": "efsw",
    "type": "static_library",
    "sources": [
      "./src/deps/efsw/src/efsw/Debug.cpp",
      ...
      "./src/deps/efsw/src/efsw/WatcherInotify.cpp"
    ],
    "include_dirs": [
      "./src/deps/efsw/include",
      "./src/deps/efsw/src"
    ],
    "conditions": [
      ["OS=="win"", {
        "sources": [
          "./src/deps/efsw/src/efsw/platform/win/FileSystemImpl.cpp",
          ...
          "./src/deps/efsw/src/efsw/WatcherWin32.cpp"
        ]
      }],
      ["OS!="win"", {
        "sources": [
          "./src/deps/efsw/src/efsw/platform/posix/FileSystemImpl.cpp",
          ...
          "./src/deps/efsw/src/efsw/WatcherKqueue.cpp"
        ]
      }],
      ["OS=="mac"", {
        "defines": [
          "EFSW_FSEVENTS_SUPPORTED"
        ],
        "xcode_settings": {
          "OTHER_LDFLAGS": [
            "-framework CoreFoundation -framework CoreServices"
          ]
        }
      }
    ]
  }]
}
```

具体的 `binding.gyp` 详情可以参照随书代码的相应目录或者 <https://github.com/XadillaX/node-efsw>。

有了这个文件关于 `efsw` 的声明之后，读者就可以尝试着编译一下了。

```
$ node-gyp rebuild
gyp info it worked if it ends with ok
gyp info using node-gyp@3.6.0
...
...
gyp info spawn make
gyp info spawn args [ 'BUILDTYPE=Release', '-C', 'build' ]
  CXX(target) Release/obj.target/efsw/src/deps/efsw/src/efsw/Debug.o
  ...
  CXX(target) Release/obj.target/efsw/src/deps/efsw/src/efsw/WatcherKqueue.o
  LIBTOOL-STATIC Release/efsw.a
gyp info ok
```

完成之后，如果是在 UNIX 系统下，读者就能在 `build/Release` 目录下发现已经编译好的静态链接库 `efsw.a` 了。

7.1.3 小结

本节初步为我们所需要实战的包 `node-efsw` 做了一个规划，代码层面做了一个分层，为大家说明了什么代码用原生 C++ 模块完成，什么代码使用 JavaScript 来完成。

然后为项目搭建了一个最初的结构，包含 `efsw` 这个第三方库的源码、`package.json` 以及 `binding.gyp`。

7.1.4 参考资料

[1] `fsevents`: <https://github.com/strongloop/fsevents>.

7.2 核心设计

本节将循序渐进地介绍 `node-efsw` 的核心部分设计，即它的 C++ 部分代码。

7.2.1 API 设计

在写代码之前，读者首先要知道我们需要什么，我们写的代码要暴露什么接口。所以，我们先开始 API 的设计吧。

首先，在 C++ 核心部分的代码中，笔者将要实现一个 JavaScript 类，这个类提供了文件监视核心逻辑的一些接口。这里姑且称它为 EFSWCore 吧。

在 7.1 节中有一张 efsw 代码层面分工的图片，即图 7-1，里面的 core.Watcher 也就是本节的 EFSWCore 中并没有什么 API，有的就是一个用于监听的 efsw 对象和一个回调函数。所以，本节中指的 API 主要就是构造函数与析构函数。

1. JavaScript 层构造函数

EFSWCore 的 JavaScript 构造函数将接收一个文件的路径，这个路径就是待监听的路径，并且再传入一个监听函数，表示这个事件被触发后要调用的函数。

2. C++ 层析构函数

在 EFSWCore 的 C++ 析构函数中，就是做一些善后的处理了，如删除一些指针释放内存，尤其是删除回调函数。

7.2.2 EFSWCore 的血肉之躯

在对 API 有了一个初步的设计之后，我们就可以开始写出这个类了。在此之前，我们先把 NAN 作为依赖加到包中：

```
$ npm install --save nan
```

然后打开 binding.gyp，把 NAN 的包含路径和一些基本信息填进去。

```
...
{
  "target_name": "core",
  "include_dirs": [
    "<!(node -e \"require('nan')\")"
  ],
  "sources": [
    "./src/entry.cc",
    "./src/efsw_core.cc"
  ],
  "dependencies": [
```

```

    "efsw"
  ],
  "conditions": [
    ["OS==" "mac\"", {
      "xcode_settings": {
        "OTHER_LDFLAGS": [
          "-framework CoreFoundation -framework CoreServices"
        ]
      }
    }]
  ]
}
...

```

1. 类的骨架

首先我们按照 5.4.4 节介绍的 NAN 如何写一个类的方式，把 EFSWCore 的骨架搭建好。

在 src 目录下新建两个文件，分别是头文件 efsw_core.h 和源文件 efsw_core.cc。打开 efsw_core.h 开始写代码。具体的写法可参照 5.4.4 节和 4.3.1 节。

```

#ifndef __EFSW_CORE_H__
#define __EFSW_CORE_H__
#include <string>
#include <nan.h>
#include "../deps/efsw/include/efsw/efsw.hpp"

namespace efsw_core {

class EFSWCore : public Nan::ObjectWrap {
public:
    static NAN_MODULE_INIT(Init);

private:
    // C++ 构造函数
    explicit EFSWCore(const char* path, Nan::Callback* listener);
    // C++ 析构函数
    ~EFSWCore();

    // JavaScript 构造函数定义
    static NAN_METHOD(New);

private:
    // 监听路径
    std::string path;

```

```

// 监听回调
Nan::Callback* listener;
// efsw 对象
efsw::FileWatcher* watcher;
};

}

#endif

```

有了这个骨架之后，打开 `efsw_core.cc` 就可以逐个实现函数了。

2. EFSWCore::New()

这个函数就是前面讲的 JavaScript 函数，如 4.3.1 节所述，这个静态函数的主要作用就是通过 `new` 分配一个 JavaScript 对象。打开 `efsw_core.cc` 开始写这个函数。

```

#include "efsw_core.h"
using namespace efsw;
using namespace v8;

namespace efsw_core {

NAN_METHOD(EFSWCore::New)
{
    Nan::Utf8String path(info[0]->ToString());
    Nan::Callback* listener = new Nan::Callback(info[1].As<v8::Function>());

    EFSWCore* core = new EFSWCore(*path, listener);
    core->Wrap(info.This());
    info.GetReturnValue().Set(info.This());
}

}

```

这段代码非常简单，由于笔者事先设定好，这个 `EFSWCore` 对象只会被笔者将要写的包的 JavaScript 类给包起来，因此各种容错处理就交给外层来做了，而不用考虑是否会通过没有 `new` 的形式来调用构造函数。

笔者设计 `new EFSWCore()` 的时候，约定有两个参数。

- ① `path`：即待监听的路径字符串。
- ② `listener`：即监听回调函数。

所以在函数的一开始，就把这两个参数抽了出来并转换为相应的数据类型。然后再声明一个 `EFSWCore` 对象指针，将 `Wrap()` 函数与 `this` 绑定在一起，最后返回这个 `this` 指针。

3. `EFSWCore::EFSWCore()` 与 `EFSWCore::~~EFSWCore()`

写完了 JavaScript 的构造函数，笔者开始写 C++ 层面的构造函数与析构函数了。这两个函数非常简单，在构造函数中只需要把传入的参数赋值给自身的一些成员变量即可，而析构函数则是释放指针。

```
EFSWCore::EFSWCore(const char* path, Nan::Callback* listener) :
    path(path),
    listener(listener)
{
    watcher = new FileWatcher();
}

EFSWCore::~~EFSWCore()
{
    Nan::HandleScope scope;
    delete watcher;
    delete listener;
}
```

4. 类模块的初始化

最后一个函数就是类模块的初始化函数了，即 `NAN_MODULE_INIT(EFSWCore::Init)`。

```
NAN_MODULE_INIT(EFSWCore::Init)
{
    v8::Local<v8::FunctionTemplate> tpl = Nan::New<v8::FunctionTemplate>(Nan::ew);
    tpl->SetClassName(Nan::New("EFSWCore").ToLocalChecked());
    tpl->InstanceTemplate()->SetInternalFieldCount(1);

    Nan::Set(target, Nan::New("EFSWCore").ToLocalChecked(),
    Nan::GetFunction(tpl).ToLocalChecked());
}
```

如果大家不是“空降”至此，而是已经阅读了 4.3.1 节，则源码几乎不需要解释。

最后，我们又顺手编写了整个模块的初始化函数。打开 `src/entry.cc` 并写入如下代码：

```
#ifndef __ENTRY_H__
#define __ENTRY_H__
```

```

#include "nan.h"
#include "efsw_core.h"

namespace efsw_core {

NAN_MODULE_INIT(Init)
{
    EFSWCore::Init(target);
}

NODE_MODULE(core, Init)

}

#endif

```

这样一来，代码就能编译通过了，只不过就目前来说，它还只是一个没有任何功能的空壳而已。

7.2.3 EFSWCore 的灵魂

搞定了 EFSWCore 的“血肉之躯”之后，笔者要开始写核心的“灵魂”逻辑——监听了。

正如 6.3.2 节所说的那样，我们先得为 EFSWCore 写一个 C++ 的 Listener 类。

1. EFSWCoreListener

新建两个文件，分别是 src/efsw_core_listener.h 和 src/efsw_core_listener.cc。先向头文件中写入监听类的骨架。

```

#ifndef __EFSW_CORE_LISTENER_H__
#define __EFSW_CORE_LISTENER_H__
#include <string>
#include <nan.h>
#include "../deps/efsw/include/efsw/efsw.hpp"

namespace efsw_core {

#define Watch watch
#define AddWatch addWatch
#define HandleFileAction handleFileAction
#define GetLastErrorLog getLastErrorLog
#define WatchId WatchID

```

```

class EFSWCoreListener : public efsw::FileWatchListener {
public:
    EFSWCoreListener(Nan::Callback* listener);
    ~EFSWCoreListener();

    void HandleFileAction(
        efsw::WatchId,
        const std::string& dir,
        const std::string& filename,
        efsw::Action action,
        std::string old_filename);

private:
    Nan::Callback* listener;
};

}

#endif

```

代码中定义了几个宏是为了让代码风格能遵循“函数名等命名以大驼峰形式”的原则。

然后在 `src/efsw_core_listener.cc` 文件中先编写出 `uv_async_t` 相关的代码。至于为什么要用 `uv_async_t` 句柄，请大家参考 6.3.1 节和 6.3.2 节中的内容。

```

#include "efsw_core_listener.h"
using namespace std;
using namespace efsw;
using namespace v8;

namespace efsw_core {

struct AsyncArgs {
    string dir;
    string filename;
    string old_filename;
    Action action;
    Nan::Callback* callback;

    uv_async_t handle;
};

void OnAsyncClosed(uv_handle_t* handle)
{

```

```

    uv_async_t* async = (uv_async_t*)handle;
    AsyncArgs* data = (AsyncArgs*)async->data;
    delete data;
}

void OnEFSWEvent(uv_async_t* handle)
{
    Nan::HandleScope scope;
    AsyncArgs* data = (AsyncArgs*)handle->data;
    Local<Value> argv[] = {
        Nan::New(data->dir.c_str()).ToLocalChecked(),
        Nan::New(data->filename.c_str()).ToLocalChecked(),
        Nan::New(data->old_filename.c_str()).ToLocalChecked(),
        Nan::New(data->action)
    };
    data->callback->Call(4, argv);
    uv_close((uv_handle_t*)handle, OnAsyncClosed);
}
}

```

在这段源码中，笔者定义了通过 `uv_async_t` 唤起的主事件循环中的函数逻辑，就是将参数中的各项内容抽出来，并通过 `data->callback->Call()` 调用来调起监听的回调函数，最后将句柄关闭。

有了这个函数之后，一切就好办了。继续向 `src/efsw_core_listener.cc` 中写入 `EFSWCoreListener` 的源码。

```

EFSWCoreListener::EFSWCoreListener(Nan::Callback* listener) :
    listener(listener)
{
}

EFSWCoreListener::~EFSWCoreListener()
{
}

void EFSWCoreListener::HandleFileAction(
    efsw::WatchId,
    const std::string& dir,
    const std::string& filename,
    efsw::Action action,
    std::string old_filename)
{
    AsyncArgs* args = new AsyncArgs();
}

```

```

args->dir = dir;
args->filename = filename;
args->old_filename = old_filename;
args->action = action;
args->callback = listener;

uv_async_init(uv_default_loop(), &args->handle, OnEFSWEvent);
args->handle.data = (void*)args;
uv_async_send(&args->handle);
}

```

在此，构造函数对传入的 `Nan::Callback*` 赋值，析构函数没有内容；而在监听回调函数 `EFSWCoreListener::HandleFileAction` 中，我们生成了一个 `AsyncArgs*` 对象并通过 `uv_async_send()` 来唤起主事件循环，以进入笔者刚才写好的函数逻辑中。

2. 在 EFSWCore 与 EFSWCoreListener 之间搭起桥梁

在完成 `EFSWCoreListener` 之后，我们要将事件监听的逻辑与 `EFSWCore` 桥接起来，主要的做法就是将 `EFSWCoreListener` 作为 `EFSWCore` 的一个成员变量存在，并在必要的时候与 `efsw::FileWatcher` 联立起来。

打开 `src/efsw_core.h`，把 `EFSWCoreListener` 和 `efsw::WatchId` 写入类的声明中。

```

...
#include "efsw_core_listener.h"

class EFSWCore ... {
...

private:
    ...

    efsw::WatchId watch_id;
    EFSWCoreListener core_listener;
};

```

接着打开 `src/efsw_core.cc`，在构造函数中把 `listener` 传入 `core_listener` 中，并通过 `watcher->AddWatch()` 开始监听目录。

```

EFSWCore::EFSWCore(const char* path, Nan::Callback* listener) :
    path(path),
    listener(listener),
    core_listener(listener)
{

```



```

watcher = new FileWatcher();

// 开始监听
watch_id = watcher->AddWatch(path, &core_listener, true);
}

```

这样一来，监听逻辑就与 EFSWCore 关联起来了。

3. 试运行

现在 EFSWCore 核心完成了，就可以开始试运行一下了。

```

$ node-gyp rebuild
...
$ node
> const EFSWCore = require("./build/Release/core").EFSWCore
undefined
> const a = new EFSWCore("/tmp", function() { console.log(arguments); });
undefined

```

当读者操作 /tmp 目录做一些事情（比如新建一个文件）的时候，就能得到相应的反馈了。这一点笔者在第 6 章中也介绍过。

```

{ '0': '/tmp/', '1': '1', '2': '', '3': 1 }
{ '0': '/tmp/', '1': '1', '2': '', '3': 3 }

```

7.2.4 小结

本节主要聚焦于设计 node-efsw 的 C++ 层面代码。由于一些容错处理交予了外部 JavaScript 代码来做，因此很大程度上我们可以忽略一些容错处理。本节代码的一些思想在前面章节中均曾涉及，如果大家一上来就阅读本书内容，并且觉得有些地方比较晦涩的话，推荐回过头去阅读先前的一些章节：

- 4.3.1 C++ 与 JavaScript 类封装；
- 5.4.4 封装一个类；
- 6.3.1 uv_async_t 句柄；
- 6.3.2 Watchdog 半成品实战解析。

7.3 编写 JavaScript 类

当我们得到了一个以 C++ 编写的模块之后，就将以一个 JavaScript 模块把它包起来，并在文档中不暴露我们的 C++ 模块接口，也不推荐使用者直接使用 C++ 模块来做事情。

7.3.1 类的设计

在编写 JavaScript 类之前，笔者先设计一下这个类。首先其名字就叫 `Watcher`，并且构造函数传入的是一个待监听的路径。

```
"use strict";

class Watcher {
  constructor(path) {
    // ...
  }
};

module.exports = Watcher;
```

新建一个 JavaScript 文件 `lib/watcher.js`，并把上面的代码复制进去，然后开始编写代码吧。

1. 继承关系

既然这是一个事件监听相关的类，那么最好的继承关系自然是将 `Watcher` 继承自 `EventEmitter` 了。将刚才写的代码稍加修饰。

```
"use strict";

const EventEmitter = require("events").EventEmitter;

class Watcher extends EventEmitter {
  constructor(path) {
    super();
    Object.defineProperty(this, {
      path: {
        value: path,
        configurable: false,
        enumerable: true,
        writable: false
      }
    });
  }
}
```

```

}

module.exports = Watcher;

```

为了类的简洁性,也为了减少开发成本,笔者设定了这个类的对象对应的路径一经设定就不能修改,所以在构造函数中通过 `Object.defineProperty()` 来将 `path` 字段设定为只读的。

2. 塞入 EFSWCore

继续修改代码,将 `EFSWCore` 引入文件中,并在构造函数中一并通过 `Object.defineProperty()` 挂到 `Watcher` 对象中。

由于本节还不涉及详细的逻辑,因此监听的回调暂时用一个示例函数代替。

```

...

const EFSWCore = require("../build/Release/core").EFSWCore;

class Watcher extends EventEmitter {
  constructor(path) {
    super();
    Object.defineProperty(this, {
      path: {
        value: path,
        configurable: false,
        enumerable: true,
        writable: false
      },
      core: {
        value: new EFSWCore(path, function() {
          console.log(arguments);
        }),
        configurable: false,
        enumerable: false,
        writable: false
      }
    });
  }
}

...

```

这样一来,这个 `Watcher` 样例就完成了。笔者顺便在项目的根目录下新建一个 `efsw.js` 文件,并将 `lib/watcher.js` 引入进来。

```
// efsw.js
module.exports.Watcher = require("../lib/watcher");
```

运行 Node.js 看看效果吧。

```
$ node
> const Watcher = require("./").Watcher;
undefined
> new Watcher("/tmp")
Watcher {
  domain:
    Domain {
      domain: null,
      _events: { error: [Function] },
      _eventsCount: 1,
      _maxListeners: undefined,
      members: [] },
  _events: {},
  _eventsCount: 0,
  _maxListeners: undefined,
  path: '/tmp' }
```

在此能看到，当我们通过 `new` 分配了一个 `Watcher()` 之后，这个对象也就生成了。在 `/tmp` 下做一些事情时，也会得到一些相应的反馈。

7.3.2 核心逻辑

在初步把 `EFSWCore` 继承到我们的 `Watcher` 之后，就可以开始写核心的逻辑了。主要的逻辑将集中在待传入 `EFSWCore` 的函数中。

1. 替换监听函数

打开 `lib/watcher.js`，我们先写一个监听函数的“皮”，里面没有任何逻辑，然后将先前在执行 `new EFSWCore()` 时传入的函数替换成新的函数。

```
class Watcher extends EventEmitter {
  constructor(path) {
    super();
    Object.defineProperties(this, {
      path: {
        value: path,
        configurable: false,
        enumerable: true,
```

```

        writable: false
      },
      core: {
        value: new EFSWCore(path, this._onEFSWEvent.bind(this)),
        configurable: false,
        enumerable: false,
        writable: false
      }
    });
  }

  _onEFSWEvent(path, filename, oldFilename, action) {
    // TODO
  }
}

```

这么一来，一旦有文件系统的事件被触发，就会执行 `_onEFSWEvent()` 函数了。

2. 事件逻辑

在前文中笔者写了一个函数的“皮”，现在要往里面塞入具体的逻辑。回想本章最初的时候，笔者假想了一下事件被触发时的参数。

```

const Watcher = require('efsw').Watcher;
const watcher = new Watcher(__dirname);
watcher.on('change', function(path, info) {
  console.log(path, info.action, info.old);
});

```

第一个参数是文件的详细路径名，然后 `info` 中有其他的一些信息。笔者在这里细化一下 `info` 中的各字段。

- `dir`: 监听的路径，该字段为字符串。
- `action`: 事件类型，该字段为字符串。
- `relative`: 相对于监听路径的文件相对路径，该字段为字符串。
- `old`: 若有，则代表原文件路径，该字段为字符串。
- `oldRelative`: 若有，则代表相对于监听路径的原文件相对路径，该字段为字符串。

这样一来，我们的 `_onEFSWEvent()` 函数职责就非常清晰了，把传进来的参数整理成 `path` 和 `info`，并触发当前事件对象的 `change` 事件。

所以，打开 `lib/watcher.js`，我们就可以继续在此修改代码了。

```

const path = require("path");

...

const ACTION_MAP = {
  "1": "ADD",
  "2": "DELETE",
  "3": "MODIFIED",
  "4": "MOVED"
};

class Watcher extends EventEmitter {
  ...

  _onEFSWEvent(_path, filename, oldFilename, action) {
    const fullPath = path.join(_path, filename);
    const info = {
      dir: _path,
      action: ACTION_MAP[action] || "UNKNOWN",
      relative: filename
    };

    if(oldFilename) {
      info.old = path.join(_path, oldFilename);
      info.oldRelative = oldFilename;
    }

    this.emit("change", fullPath, info);
  }
}

...

```

3. 初步运行

基本的代码到此就完成了，那么接下来就可以运行一遍看看了。

```

$ node
> const Watcher = require("./").Watcher;
undefined
> const a = new Watcher("/tmp");
undefined
> a.on("change", function() { console.log(arguments); });
Watcher {
  domain:

```

```
Domain {
  domain: null,
  _events: { error: [Function] },
  _eventsCount: 1,
  _maxListeners: undefined,
  members: [] },
_events: { change: [Function] },
_eventsCount: 1,
_maxListeners: undefined,
path: '/tmp' }
```

当读者在 /tmp 目录中做一些事情的时候，就能得到相应的反馈了。

```
{ '0': '/tmp/2.4',
  '1':
  { dir: '/tmp/',
    action: 'MOVED',
    relative: '2.4',
    old: '/private/tmp/2.3',
    oldRelative: '2.3' } }
```

7.3.3 简单容错

在基本的逻辑完成之后，读者要进一步考虑问题了，如一些容错的处理。虽然这应该是 Node.js 初学的时候在设计函数时就应该掌握的技能，但是本书在这里也稍微实现一下这些逻辑。

1. 相对路径

如果传入构造函数的路径是一个相对路径，那么笔者推荐将其转换成一个绝对路径之后再传入 EFSWCore 对象中（虽然 efsw 库也支持相对路径）。这个事情在 lib/watcher.js 的构造函数中完成。

```
constructor(_path) {
  _path = path.resolve(process.cwd(), _path);

  super();

  // 注意下方的 `path` 全改成了 `_path`
  Object.defineProperties(this, {
    path: {
      value: _path,
      configurable: false,
      enumerable: true,
```

```

        writable: false
    },
    core: {
        value: new EFSWCore(_path, this._onEFSWEvent.bind(this)),
        configurable: false,
        enumerable: false,
        writable: false
    }
});
}

```

2. 错误路径或者路径不存在

当新建一个 `Watcher` 对象的时候，若路径错误，或者不存在，就抛出一个错误。这个事情也是在 `lib/watcher.js` 的构造函数中完成的。

```

const fs = require("fs");

class Watcher extends EventEmitter {
    constructor(_path) {
        _path = path.resolve(process.cwd(), _path);

        super();
        ...

        const self = this;
        fs.stat(_path, function(err, stat) {
            if(err) return self.emit("error", err);

            // 若监听路径不是文件夹或不存在，则触发 error 事件
            if(!stat.isDirectory()) {
                return self.emit(
                    "error",
                    new Error(`${_path} is not a directory.`));
            }
        });
    }
}

```

3. 事件类型不对

当我们的监听函数得到的 `action` 不属于 1、2、3、4 中的任意一项时，那就是事件类型不对了，当然这种错误基本上不会发生。所以，其实容错不做也没关系，若读者有兴趣的话也可以加一下。仍然是 `lib/watcher.js`，只不过是 `_onEFSWEvent()` 函数。


```

_onEFSWEvent(_path, filename, oldFilename, action) {
    ...

    if(info.action === "UNKNOWN") {
        const err = new Error(`Unknown fs event ${action}.`);
        err.info = info;
        this.emit("error", err);
    } else {
        this.emit("change", fullPath, info);
    }
}
}

```

7.3.4 小结

本节初步完成了 `node-efsw` 这个包的基本功能，并且有了一定的健壮性。这是一个使用 C++ 模块作为内核，并在上层使用 Node.js 来铸造一个包的典型方法：

内核做主要的核心逻辑，不需要处理太多输入 / 输出上的容错处理，且不暴露给开发者；而 Node.js 层面做的主要是一些输入 / 输出的容错处理，保证将比较正确的数据传入 C++ 模块中。

7.4 进一步完善

在 7.2 节和 7.3 节中，笔者完成了一个最简易的文件监视的包。但是与 `fsevents` 相比，它还缺少一个非常重要的功能，那就是启动和停止监听。

```

watcher.start(); // 开始监听
watcher.stop();  // 停止监听

```

笔者之前完成的功能是，一旦初始化了一个对象，监听的操作就开始了，无论你有没有增加 `.on()` 的事件回调，都无法停止它。所以，本节内容的完善主要是完成这两个函数。

7.4.1 C++ 代码的完善

要实现 `.start()` 和 `.stop()` 这两个函数，我们要在 C++ 层面就做好这件事情。做法很简单：这就是在 `ESFWCore` 的构造函数中，笔者本来是直接添加了事件的监听，在完善后，我们将这一步移到 `ESFWCore::Start()` 和 `ESFWCore::End()` 中去做。

1. 类声明完善

打开 `src/efsw_core.h`，在类声明中加上几个函数的声明，如下：

```
class EFSWCore : public Nan::ObjectWrap {
public:
    static NAN_MODULE_INIT(Init);

private:
    explicit EFSWCore(const char* path, Nan::Callback* listener);
    ~EFSWCore();

    efsw::WatchId Start();
    void Stop();

    static NAN_METHOD(New);
    static NAN_METHOD(Start);
    static NAN_METHOD(Stop);

    ...
};
```

在此能看到，在 `EFSWCore` 本身的成员函数中多了 `Start()` 和 `Stop()`，而同时又多了供 JavaScript 层面调用的两个静态函数——`NAN_METHOD(Start)` 和 `NAN_METHOD(Stop)`。

2. WatchId EFSWCore::Start()

下面首先完成 `Start()` 函数。打开 `src/efsw_core.cc`，先把构造函数改好——移除 `watcher->AddWatch()` 以及初始化 `watch_id`，就像这样：

```
EFSWCore::EFSWCore(const char* path, Nan::Callback* listener) :
    path(path),
    listener(listener),
    core_listener(listener)
{
    watcher = new FileWatcher();
    watch_id = 0;
}
```

然后在 `EFSWCore::Start()` 中添加监听，并且写入 `watch_id` 及返回。

```
WatchId EFSWCore::Start()
{
    if(watch_id > 0)
    {
```

```

        return -100;
    }

    WatchId ret = watcher->AddWatch(path.c_str(), &core_listener, true);
    if(ret > 0) watch_id = ret;

    return ret;
}

```

在这个函数中需要注意以下几点：首先就是，如果当前的 `EFSWCore` 已经处于监听状态，则直接返回一个异常码（如 `-100`），否则我们就新增一个监听。但若监听的结果是失败的，则我们就不对 `watch_id` 进行赋值。最后，我们返回监听结果就可以了。

3. `void EFSWCore::Stop()`

在 `Stop()` 函数中，我们只需要判断 `watch_id` 是否合法。若合法，则将其监听移除即可。

```

void EFSWCore::Stop()
{
    if(watch_id <= 0) return;

    watcher->RemoveWatch(watch_id);
    watch_id = 0;
}

```

最后，别忘了在相应的地方进行一个宏定义，把小驼峰形式变成大驼峰形式。

```
#define RemoveWatch removeWatch
```

4. `NAN_METHOD(Start)`

在 C++ 层面提供了 `Start()` 的接口，我们要将其暴露到 JavaScript 层面。

```

NAN_METHOD(EFSWCore::Start)
{
    EFSWCore* core = Nan::ObjectWrap::Unwrap<EFSWCore>(info.Holder());
    WatchId watch_id = core->Start();

    if(watch_id < 0)
    {
        if(watch_id == -100)
        {
            return Nan::ThrowError("Watcher is already in watching.");
        }
    }
}

```

```

        return Nan::ThrowError(Errors::Log::GetLastErrorLog().c_str());
    }
}

```

在代码中，第一行先通过 `ObjectWrap()` 把 `info.Holder()` 即 JavaScript 的当前对象中隐匿的 `EFSWCore` “揪”出来，然后在第二行执行 `core->Start()` 开始监听。

这时我们能得到一个监听结果，做一个判断。若结果小于 0，则代表监听失败，那么抛出相应的错误。其中 -100 是笔者事先定义好的“已处于监听状态”的错误，其他错误是来自 `efsw` 本身的错误，通过 `Errors::Log::GetLastErrorLog().c_str()` 获取错误详细信息。

5. NAN_METHOD(Stop)

`Stop` 函数就更简单了，除了外面声明的内容，一共只有两行代码，也不用怎么解释了。

```

NAN_METHOD(EFSWCore::Stop)
{
    EFSWCore* core = Nan::ObjectWrap::Unwrap<EFSWCore>(info.Holder());
    core->Stop();
}

```

6. 把函数接到原型链上

我们光把函数写好没有什么用，还需要把它们接到原型链上才能真正地使用。所以修改一下 `NAN_MODULE_INIT(EFSWCore::Init)`，把原型链上的内容拼上去吧。

```

NAN_MODULE_INIT(EFSWCore::Init)
{
    v8::Local<v8::FunctionTemplate> tpl = Nan::New<v8::FunctionTemplate>(Nan::New("EFSWCore").ToLocalChecked());
    tpl->SetClassName(Nan::New("EFSWCore").ToLocalChecked());
    tpl->InstanceTemplate()->SetInternalFieldCount(1);

    Nan::SetPrototypeMethod(tpl, "start", Start);
    Nan::SetPrototypeMethod(tpl, "stop", Stop);

    Nan::Set(target, Nan::New("EFSWCore").ToLocalChecked(),
    Nan::GetFunction(tpl).ToLocalChecked());
}

```

7. 试运行

当前笔者已经把 EFSWCore 改造完成了，可以尝试编译并执行 Node.js 试验一下了。

```
$ node-gyp rebuild
...
$ node
> const Watcher = require("./build/Release/core").EFSWCore;
undefined
> const a = new Watcher("/tmp", function() { console.log(arguments); });
undefined
```

当你执行了这些代码之后，去 /tmp 下做一些相应的事情，由于此时还未开始监听事件，因此不会有任何反馈。接下来继续执行一行代码。

```
> a.start();
undefined
```

这时读者去 /tmp 下“涂个鸦”，事件就被触发了，这样终端会有相应的输出。

然后当读者继续执行 a.start() 时，终端会抛出错误：

```
Error: Watcher is already in watching.
    at Error (native)
    at repl:1:3
    at sigintHandlersWrap (vm.js:96:12)
    at REPLServer.defaultEval (repl.js:313:29)
    at bound (domain.js:280:14)
    at REPLServer.<anonymous> (repl.js:513:10)
    at emitOne (events.js:101:20)
    at REPLServer.emit (events.js:188:7)
    at REPLServer.Interface._onLine (readline.js:239:10)
    at REPLServer.Interface._line (readline.js:585:8)
```

接下来当读者再执行 a.stop() 时，再次跑到 /tmp 目录下做事情，事件又不会被触发了。基本上事态已经按照我们的预想在进行了。

并且在我们新建一个 Watcher 对象时，传入的若是一个不存在的路径，则相应的 efs 错误也会被抛出来，例如：

```
Error: File not found ( /tmp/safd/sadf )
    at Error (native)
    at repl:1:3
    at sigintHandlersWrap (vm.js:96:12)
```

```

at REPLServer.defaultEval (repl.js:313:29)
at bound (domain.js:280:14)
at REPLServer.<anonymous> (repl.js:513:10)
at emitOne (events.js:101:20)
at REPLServer.emit (events.js:188:7)
at REPLServer.Interface._onLine (readline.js:239:10)
at REPLServer.Interface._line (readline.js:585:8)

```

7.4.2 JavaScript 代码的完善

我们既然完善了 C++ 层面的代码，那么相应地还应该完善 JavaScript 层面的代码。打开 lib/watcher.js，开始进行相应的改造吧。

1. 构造函数的改造

既然我们已经有了监控中和非监控中的状态，那么设计的时候就应该有一个变量来标明当前的状态——在构造函数中声明默认值为 false。

并且，由于我们在 EFSWCore::Start() 中已经有了一些基础的监听容错处理了，因此构造函数中的 fs.stat 就不再需要了。所以整合下来，Watcher 的构造函数就变成了如下这样：

```

constructor(_path) {
  _path = path.resolve(process.cwd(), _path);

  super();
  Object.defineProperties(this, {
    path: {
      value: _path,
      configurable: false,
      enumerable: true,
      writable: false
    },
    core: {
      value: new EFSWCore(_path, this._onEFSWEvent.bind(this)),
      configurable: false,
      enumerable: false,
      writable: false
    }
  });
  this.watching = false;
}

```

2. start()

Watcher 的 `start()` 函数很简单，直接调用 `core` 的 `start()` 并捕获错误进行处理就可以了，顺手标记一下 `this.watching`。

```
start() {
  if(this.watching) {
    return this.emit("error", new Error("Watcher is already in
watching."));
  }

  try {
    this.core.start();
  } catch(e) {
    return this.emit("error", e);
  }
  this.watching = true;
}
```

在这段代码中，首先判断一下 `this.watching`，看看是不是重复监听了。若是，则触发一个错误事件，否则就开始监听。在调用 `this.core.start()` 的过程中，若内核抛出错误，就触发一个错误事件，否则标记 `this.watching` 为 `true`。

3. stop()

Watcher 的 `stop()` 函数就更简单了，只有两步：

- ① 调用 `this.core.stop()`；
- ② 标记 `this.watching` 为 `false`。

```
stop() {
  this.core.stop();
  this.watching = false;
}
```

4. 试运行

改造也大功告成了，继续执行一遍 Node.js 来试验吧。

```
$ node
> const Watcher = require("./").Watcher;
undefined
> const a = new Watcher("/tmp");
undefined
```

```
> a.on("change", function() { console.log(arguments); });
...
> a.start();
undefined
> a.start();
false

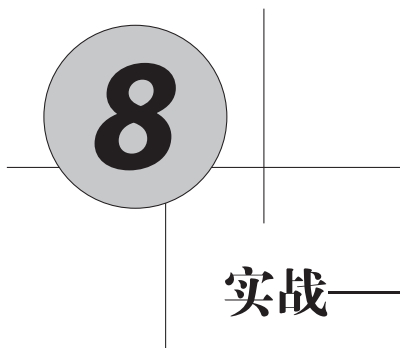
# 触发错误事件
Error: Watcher is already in watching.
    at Watcher.start (/.../35. efsw advanced/lib/watcher.js:45:39)
    at repl:1:3
    at sigintHandlersWrap (vm.js:96:12)
    at REPLServer.defaultEval (repl.js:313:29)
    at bound (domain.js:280:14)
    at REPLServer.<anonymous> (repl.js:513:10)
    at emitOne (events.js:101:20)
    at REPLServer.emit (events.js:188:7)
    at REPLServer.Interface._onLine (readline.js:239:10)
    at REPLServer.Interface._line (readline.js:585:8)
> a.stop();
undefined
```

总之按照自己的想法，想怎么试验就怎么试验吧，程序都是能按照自己预想的路线执行下去的。

7.4.3 小结

本节帮助大家完善了 `node-efsw` 这个包，使其能够按照开发者自身的意愿随时去启动或者停止对于路径的监听。这样的包基本上已经处于一个可发布的状态了，再增加一些单元测试和文档，这就是一个完整的包了。

本章内容到此就结束了。至此，读者应该对于 Node.js 的 C++ 原生包的开发有了一个实战性的认知了。



实战——现有包剖析

在第 7 章中，笔者介绍了写一个 C++ 包的通用做法，并完成了一个文件监视器的包。在本章中笔者将再为大家举几个 Node.js 的 C++ 扩展的例子，从源码层面做一个简要的剖析，让大家对 C++ 扩展有进一步的印象。

8.1 字符串哈希模块——Bling Hashes

参加过 ACM 竞赛的读者对此会比较熟悉：当我们要对某个字符串做哈希的时候，会用到一些哈希算法，如 BKDRHash、APHash 等。

关于将这些哈希算法封装起来给 Node.js 用的一些情况，笔者在 2.1.2 节中曾介绍过，本节将继续展开来讲。

可能大家参考本包的源码仓库（https://github.com/XadillaX/bling_hashes）一起阅读本节内容，效果会更佳。

8.1.1 文件设定

1. 基本哈希函数文件

在写这个包之前，笔者内心的设想非常明确，就是要实现一套这些基础哈希¹算法的接口，而这些实现的代码都是直接从 ByVoid 博客的整理中复制、粘贴的——这都是一代人的结晶。

首先，为这些算法新建一个头文件和一个源文件，分别是 `src/byvoid/algorithms.h` 和 `src/byvoid/algorithms.cxx`。

其次，关于这些基础字符串哈希的 C++ 模块 API 调用方式如下：

```
// 表示调用 DJBHash
const hash = binding.calcHash("DJB", "Hello world!");
```

所以，还需要新建一个 `src/byvoid/entry.h` 来实现这个 `calcHash()` 函数。

2. CityHash

自 Bling Hashes 后续的一个版本开始，笔者决定加入谷歌推出的一个哈希算法——CityHash²。由于 CityHash 的算法复杂度较高，且有着比较多的基于整型数据的位运算操作，因此把 C++ 源码“扒”到 JavaScript 中的成本较高，还是直接将其 C++ 源码进行封装比较容易。

既然是封装，那就直接把 CityHash 的源码加入笔者的项目中。笔者除了直接使用 Git 的 Submodule 形式外，也能直接把文件放到项目中——CityHash 的源码文件（<https://github.com/google/cityhash/tree/master/src>）数量较少，有用的只有 3 个，分别是 `city.cc`、`city.h` 以及 `citycrc.h`，笔者将这 3 个文件加入到项目的 `src/cityhash/` 目录下。

同时，为了封装起来暴露给 JavaScript 使用，笔者还在 `src/cityhash/` 目录下新建了一个 `entry.h` 文件。

3. 其他文件

剩下的就是一些基础类的文件了。

- **index.js**: JavaScript 层面的封装。
- **binding.gyp**: GYP 文件。
- **src/bling.cxx**: 模块初始化（入口）文件。

¹ <https://www.byvoid.com/zhs/blog/string-hash-compare>

² <https://github.com/google/cityhash>

8.1.2 C++ 源码剖析

在了解了 bling hashes 这个包的目录结构之后，笔者就可以开始剖析各源码部分了。

1. binding.gyp

在本项目的 binding.gyp 当中，只需要定义 NAN 的包含路径和 3 个源文件的路径即可。

由于偷懒，因此对于基础哈希函数和 CityHash 函数的接口文件 entry，笔者只写了头文件，并没有加源文件，函数的实现直接写到了头文件中。

```
{
  "targets": [
    {
      "target_name": "bling",
      "sources": [
        "./src/byvoid/algorithms.cxx",
        "./src/cityhash/cityhash.cxx",
        "./src/bling.cxx"
      ],
      "include_dirs": [
        "<!(node -e \"require('nan')\")"
      ]
    }
  ]
}
```

2. 基础哈希函数

笔者的 Bling Hashes 做的就是封装的工作。所以，先把前人积累下来的代码放到源码文件和头文件中，也就是把 ByVoid 博客中提到的各种基础字符串哈希函数加入 src/byvoid/algorithms.cxx 和 src/byvoid/algorithms.h 中。

src/byvoid/algorithms.h:

```
#ifndef __ALGORITHMS_H__
#define __ALGORITHMS_H__

unsigned int SDBMHash(char *str);
unsigned int RSHHash(char *str);
unsigned int JSHHash(char *str);
unsigned int PJWHash(char *str);
unsigned int ELFHash(char *str);
unsigned int BKDRHash(char *str);
```

```

unsigned int DJBHash(char *str);
unsigned int APHash(char *str);

typedef unsigned int (*ByvoidHashFunc)(char*);

#endif

```

src/byvoid/algorithms.cxx:

```

#include "algorithms.h"

/**
 * 该文件参考自 https://www.byvoid.com/blog/string-hash-compare/
 */

...

// 篇幅所限，此处就放出一个函数，读者可自行到项目中查看其余函数
unsigned int RSHash(char *str)
{
    unsigned int b = 378551;
    unsigned int a = 63689;
    unsigned int hash = 0;

    while (*str)
    {
        hash = hash * a + (*str++);
        a *= b;
    }

    return (hash & 0x7FFFFFFF);
}

...

```

有了这些基础函数后，就可以写前面说的 `calcHash()` 函数了。具体实现在 `src/byvoid/entry.h` 中：

```

void ToUpperCase(std::string& str)
{
    for(unsigned int i = 0; i < str.length(); i++)
    {
        if(str[i] >= 'a' && str[i] <= 'z')
        {
            str[i] = str[i] - 'a' + 'A';
        }
    }
}

```

```

    }
}

NAN_METHOD(CalcHash)
{
    // argument length...
    if(info.Length() < 2)
    {
        return Nan::ThrowError("invalid argument count");
    }

    String::Utf8Value v8_algorithm_type(info[0]->ToString());
    String::Utf8Value v8_string(info[1]->ToString());

    // type to uppercase
    std::string algorithm_type = *v8_algorithm_type;
    ToUpperCase(algorithm_type);

    ByVoidHashFunc func = BKDRHash;
    if(algorithm_type == "SDBM") func = SDBMHash;
    else
    if(algorithm_type == "RS") func = RSHash;
    else
    if(algorithm_type == "JS") func = JSHash;
    else
    if(algorithm_type == "PJW") func = PJWHash;
    else
    if(algorithm_type == "ELF") func = ELFHash;
    else
    if(algorithm_type == "BKDR") func = BKDRHash;
    else
    if(algorithm_type == "DJB") func = DJBHash;
    else
    if(algorithm_type == "AP") func = APHash;

    unsigned int hash = func(*v8_string);

    info.GetReturnValue().Set(hash);
}

```

在 `CalcHash()` 中，一开始先判断参数个数是否为 2，若不是则抛出异常。第一个参数代表需要用到算法，第二个参数就是待计算的字符串了。

然后代表算法的字符串进行一遍大写化之后开始判断是什么算法，在得到了算法名后，将 `ByvoidHashFunc func` 这个变量赋值成对应的哈希函数的¹。

最后执行这个函数，也就是 `func(*v8_string)` 后，我们就得到了一个无符号整型的哈希值，返回即可。

3. CityHash

CityHash 的核心逻辑直接用了已有的代码，并且将其复制到 `src/cityhash/` 目录下，最后笔者只需要实现 `entry.h` 中的逻辑就可以了。

函数分两段，分别是 CityHash32 和 CityHash64，即 32 位无符号整型和 64 位无符号整型的结果。

```
NAN_METHOD(_CityHash32)
{
    if(info.Length() < 1)
    {
        return Nan::ThrowError("invalid argument count");
    }

    String::Utf8Value v8_source_string(info[0]->ToString());
    std::string source_string = *v8_source_string;
    unsigned int len = source_string.size();

    unsigned int hash = CityHash32(source_string.c_str(), len);

    info.GetReturnValue().Set(hash);
}
```

在 `_CityHash32()` 函数中，在判断了参数等内容之后，通过调用 CityHash 库的 `CityHash32()` 函数得到无符号整型的哈希值，返回即可。

下面是 `_CityHash64()` 的源码。

```
void __CityHash64FreeCallback(char* data, void* hint)
{
    unsigned long long* hash = (unsigned long long*)data;
    delete hash;
}

NAN_METHOD(_CityHash64)
```

¹ 在 `src/byvoid/algorithms.h` 中有 `typedef unsigned int (*ByvoidHashFunc)(char*);` 这样一段定义。

```

{
    if(info.Length() < 1)
    {
        return Nan::ThrowError("invalid argument count");
    }

    String::Utf8Value v8_source_string(info[0]->ToString());
    std::string source_string = *v8_source_string;
    unsigned int len = source_string.size();

    unsigned long long* hash = new unsigned long long(CityHash64(source_
string.c_str(), len));
    info.GetReturnValue().Set(
        Nan::NewBuffer(
            (char*)hash,
            sizeof(unsigned long long),
            __CityHash64FreeCallback,
            NULL).ToLocalChecked());
}

```

在函数中，前面的逻辑跟 `_CityHash32()` 一样，就是在计算哈希值的时候，笔者声明了一个 `unsigned long long*` 指针，并通过 `CityHash64()` 函数将计算出来的结果赋值到新声明出来的堆内存中，即刚才说的指针——这就形成了一个 Buffer 块。

在返回结果的时候，笔者通过 `Nan::NewBuffer()` 来生成一个 Node.js 的 Buffer，并传入刚生成的 `unsigned long long*` 指针作为 Buffer 的内容体。`Nan::NewBuffer()` 的第 3 个参数是在垃圾回收的时候用于回收刚才“new”出来的内存块 `unsigned long long*` 的，传的就是事先写好的 `__CityHash64FreeCallback`，在这个函数中做的事情就是删除这个指针释放内存，即 `delete hash`。

4. C++ 入口文件

在这个项目中，C++ 代码最简单的就是 C++ 入口文件了，即 `src/bling.cxx`。

```

NAN_MODULE_INIT(InitAll)
{
    // BYVoid hashes
    Set(target, New<String>("calcHash").ToLocalChecked(),
        GetFunction(New<FunctionTemplate>(CalcHash)).ToLocalChecked());

    // City hashes
    Set(target, New<String>("cityHash32").ToLocalChecked(),
        GetFunction(New<FunctionTemplate>(_CityHash32)).

```

```

ToLocalChecked());
    Set(target, New<String>("cityHash64").ToLocalChecked(),
          GetFunction(New<FunctionTemplate>(_CityHash64)).
    ToLocalChecked());
}

NODE_MODULE(bling, InitAll)

```

把几个函数凑到这个 `InitAll()` 里面导出一遍就可以了。

8.1.3 JavaScript 源码剖析

在 Bling Hashes 的文档中，关于基础哈希函数的写法是这么定的：

```
var hash = bling.bkdr("Hello world!");
```

但是在 C++ 代码中并没有暴露出这样的函数接口，所以需要在外层的 JavaScript 代码中封装一下，也就是 `index.js`。

1. 基础哈希函数

笔者在 C++ 的代码中暴露出来的基础哈希计算函数是这样的：

```
binding.calcHash(" 算法 ", " 待计算字符串 ");
```

而 JavaScript 层暴露出去的函数则如下：

```
bling.bkdr(" 待计算字符串 ");
```

所谓的封装无非是写好 `bling.bkdr()` 等函数，并且它们的实际逻辑其实就是返回 `binding.calcHash()` 的结果。例如：

```

var bling = require("../build/Release/bling");

exports.bkdr = function(str) {
    return bling.calcHash("BKDR", str);
};

exports.djb = function(str) {
    return bling.calcHash("DJB", str);
};

...

```


2. CityHash

CityHash 的 32 位函数封装比较简单，直接返回就可以了：

```
exports.city32 = function(str) {  
  return bling.cityHash32(str);  
};
```

但是 64 位的函数封装就复杂一些。由于 Node.js 所依赖的 ECMAScript 并不支持 64 位整数，因此要借助一些第三方库，如 long¹。

关于 long 的具体 API 文档，大家可自行查询它在 NPM 上的页面。笔者在代码中用到的就是它的构造函数 new Long(low, high, unsigned)，其有 3 个参数：

- low：低位 32 位整数。
- high：高位 32 位整数。
- unsigned：是否无符号。

大家是否还记得，笔者在 C++ 代码的 _CityHash64() 函数中返回的是一个实体为 unsigned long long 的 Buffer，所以我们在 JavaScript 封装中拿到这个 Buffer 的时候，从中分别读取高位和低位的两个 32 位整数传入 long 就可以了。

```
exports.city64 = function(str) {  
  var _int64 = bling.cityHash64(str);  
  _int64 = new Long(_int64.readInt32LE(0), _int64.readInt32LE(4), true);  
  return _int64;  
};
```

也就是说，这个 city64() 函数最后返回的是一个 Long 对象，其代表值就是计算出来的 64 位无符号整数。

8.1.4 小结

在本节中，笔者举了 bling-hashes 这个包的例子，给大家稍微温习了一些知识点，并让大家看到了 Buffer 在 C++ 模块中的一个实战用途，同时也以比较小的成本完成了各种整数计算的字符串哈希函数。若这些都移交给 JavaScript 来做，工作量会大很多。

¹ <https://www.npmjs.com/package/long>

8.1.5 参考资料

- [1] General Purpose Hash Function Algorithms: <http://www.partow.net/programming/hashfunctions/#AvailableHashFunctions>.
- [2] 各种字符串 Hash 函数比较: <https://www.byvoid.com/zhs/blog/string-hash-compare>.
- [3] More Hash Function Tests: <http://aras-p.info/blog/2016/08/09/More-Hash-Function-Tests/>.

8.2 类 Proxy 包——Auto Object

在 ECMAScript 中, Proxy 是一种特殊的类。本节主要就是为了实现与 Proxy 原理类似的包,但是用法却很不一样。

Auto Object 这个包的源码仓库在 <https://github.com/XadillaX/auto-object> 中,大家在阅读本节内容的时候可以配合源码仓库一起看。

8.2.1 Proxy

1. 基本概念

Proxy 对象用于定义基本操作的自定义行为(例如属性查找、赋值、枚举、函数调用等)。

在以下简单的例子中,当对象中不存在属性名时,默认返回数为 37。在例子中使用了 get 处理器(get handler)。¹

```
let handler = {
  get: function(target, name){
    return name in target ? target[name] : 37;
  }
};

let p = new Proxy({}, handler);

p.a = 1;
p.b = undefined;

console.log(p.a, p.b);
```

¹ 这两段内容以及下面的一段代码均取自 https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Proxy。

```
// 1 undefined

console.log('c' in p, p.c);
// false 37
```

Proxy 是一个类，其构造函数接收两个参数：第一个参数就是将要被 Proxy 包装的对象，第二个参数是一个处理代理行为的函数，在被包装对象通过某个字段访问其属性的时候会被调用。

在上面的代码中，笔者定义了第二个参数 handler 的 getter 函数，里面的逻辑是经包装后的 Proxy 对象中是否存在当前待访问的字段。若有，则返回该字段所对应的内容，否则返回 37。待包装的对象是一个空对象，所以最后这个 p 就是经 Proxy 包装后的空对象。

接着笔者对 p 挂载上 a 和 b 两个字段，其值分别为 1 和 undefined。

由于 p.a 和 p.b 已经被赋值了，说明字段存在，因此在第一个 console.log 中自然就输出了 1 undefined。而之后的一个输出中，'c' in p 的值显然是 false。既然 p.c 不存在，那么 p.c 经由 handler 处理之后的结果自然就是 37 了。

2. 应用举例

在笔者所在公司的内部，用了一套基于 Dubbo¹ 深度定制的 RPC² 服务框架。Node.js 要访问这些 Java 服务的 RPC 函数是通过定制的 HTTP 协议来完成的，所有的 RPC 服务节点都到 Zookeeper³ 进行注册。

笔者的 Node.js 团队有一个专门的私有包 Dusbo，用于调用这些 RPC 服务。其用法类似于这样：

```
const dusbo = new Dusbo("com.souche.car...");

dusbo.match({
  brandName: "雪弗兰",
  ...
}, 1000, function(err, ret, node) {
  ...
});
dusbo.matchBrand(...);
dusbo.matchSeries(...);
```

1 <http://dubbo.io/>

2 Remote Procedure Call，远程过程调用。

3 <http://zookeeper.apache.org/>

正如代码中所示，调用方法就是 `dusbo. 远程函数名 (远程参数, [超时时间,] 回调函数)`。然而，实际上每个服务的远程函数名都不一样，我们通过 `new` 分配了一个 `Dusbo` 对象的时候，唯一的不同就是构造函数传进去的包名（服务名），根本没地方自行加上各种函数。就算加上去了，函数那么多，维护一个这样的列表也是一项比较耗费人力成本的活儿——而且我们还要保持列表一致。

实际上 `Dusbo` 还有一个最终调用的函数，叫 `send()`：

```
class Dusbo {
  ...

  send(method, params, timeout, callback) {
    // 真正的调用逻辑
  }
}
```

也就是说，我们除了用上面的方法调起一个 `RPC` 函数外，还能这么调用：

```
dusbo.send("match", { ... }, 1000, function(err, ret, node) {});
dusbo.send("matchBrand", ...);
dusbo.send("matchSeries", ...);
```

这个时候，`Proxy` 就派上用场了。如果没有用笔者自己实现的 `Auto Object` 的模块，我们在 `dusbo` 实例化出来之后建一个 `Proxy` 的处理器就可以了：

```
const _dusbo = new Dusbo("com.souche.car...");
const handler = {
  get: function(target, name) {
    return target.send.bind(target, name);
  }
};
const dusbo = new Proxy(_dusbo, handler);
```

这样一来，当访问 `dusbo.match` 的时候，实际上就是访问 `_dusbo.send.bind(_dusbo, "match")`，也就是说发起调用 `dusbo.match(...)` 的时候，实际上就是执行 `(_dusbo.send.bind(_dusbo, "match"))(...)`。这就达到了我们想要的结果——这其实就起到了一个拦截器的效果。

8.2.2 Auto Object 使用范例

前文讲述了 `Proxy` 的一些特性和用处，本节就开始讲述 `Auto Object` 了。

Auto Object 能在类上面做手脚，能生成一个基本的 AutoObject 类，它就自带了拦截器的特性了。我们能操作这个类的原型链，使其作为我们自己的类，也可以再另写一个类继承自它。还有一个不同就是，Proxy 是对一个原有的对象进行包装，返回一个新对象；而 Auto Object 的应用场景不一样，它是通过特有的函数来直接生成一个带有拦截效果的对象或者类。

1. 生成一个带拦截效果的对象

Auto Object 有一个创建特殊对象的 API：

```
const AutoObject = require("auto-object");
const obj = AutoObject.createObject(handler);
```

其中 handler 是一个函数，用于产生拦截效果，其作用类似于 Proxy 构造函数的 handler，只不过在 AutoObject.createObject() 中的 handler 是一个函数，例如：

```
const obj = AutoObject.createObject(function(name) {
  return name.toUpperCase();
});
```

这是一个把任何访问都变成大写字符串的拦截器，它的效果如下：

```
console.log(obj.test);           // TEST
console.log(obj.helloWorld);    // HELLOWORLD
console.log(obj.蛋花汤);        // 蛋花汤
```

不过这个包还自带一个特性，如果该对象中自带某些属性的话，自带属性的优先级会比拦截器高，其并不会被拦截。也就是说，如果还是上面的这个 obj，那么它会有这样的一些行为：

```
obj.test = "南瓜饼";
console.log(obj.test);           // 南瓜饼
console.log(obj.helloWorld);    // HELLOWORLD
```

AutoObject 对象的拦截器所对应要调用的函数实际上被挂在对象中的 \$\$access 的属性下。也就是说，我们能随时改变其行为（虽然并不推荐这么做）。

```
obj.$$access = function() {
  return undefined;
};

console.log(obj.test);           // undefined
console.log(obj.helloWorld);    // undefined
```

2. 写一个带拦截效果的类

我们平常写一个类（原型链写法）是这样的：

```
const Cls = function(...) {  
  ...  
};  
  
Cls.prototype.foo = function() {  
  ...  
};
```

但如果要基于 `Auto Object` 写一个带拦截效果的类，就是这样的：

```
const Cls = AutoObject.createClass("类名");  
  
Cls.prototype.$$constructor = function(...) {  
  ...  
};  
  
Cls.prototype.foo = function() {  
  ...  
};  
  
Cls.prototype.$$access = function(name) {  
  ...  
};
```

其中，由于通过函数创建一个类的时候，无法指定其构造函数，因此在 `Auto Object` 中使用原型链上定义一个 `$$constructor` 函数作为其构造函数，它会在类实例化时的构造函数中被调用。而拦截器本身，与 `Auto Object` 创建一个对象类似，就是在原型链上定义一个 `$$access` 函数。

与其他类一样，通过 `Auto Object` 创建出来的类一样能继承或者被继承。这里是一个将 `Auto Object` 生成的类继承自 `Node.js` 事件类（`EventEmitter`）的范例：

```
const EventEmitter = require("events").EventEmitter;  
const util = require("util");  
  
const Cls = AutoObject.createClass("Inherit");  
  
Cls.prototype.$$constructor = function() {  
  EventEmitter.call(this);  
};
```

```
util.inherits(Cls, EventEmitter);

Cls.prototype.foo = function() {
  ...
};
```

实际上，笔者所在公司内部 **Dusbo** 就是基于 **Auto Object** 来写的 **Dusbo** 类：

```
const Dusbo = AutoObject.createClass("Dusbo");

Dusbo.prototype.$$constructor = function(serviceName, options) {
  EventEmitter.call(this);
  ...
};

util.inherit(Dusbo, EventEmitter);

Dusbo.prototype.send = function(method, params, timeout, callback) {
  // 真正的RPC调用逻辑
  ...
};

Dusbo.prototype.$$access = function(name) {
  this[name] = this.send.bind(this, name);
  return this[name];
};

...
```

代码中的 `$$access` 函数做了一个类似懒加载（**Lazy Load**）的事情，就是比如一个 `dusbo` 对象当第一次访问某个函数（如 `match`）的时候它是不存在的，就会走入 `$$access()` 的流程；在这个流程中，它设定了 `dusbo.match = this.send.bind(this, "match")`，然后再返回这个被 `.bind()` 函数进行绑定的函数。下一次再有代码要访问 `dusbo.match` 的时候，这个属性已经存在并且被赋值成了 `this.send.bind(this, "match")`，优先级高于 `$$access` 函数。于是就不会走到 `$$access` 的流程而是直接得到这个被绑定的 `send` 函数，这样就不用每次访问同一个属性的时候都新绑定一个函数返回。

8.2.3 代码剖析

在了解了 **Proxy** 以及 **Auto Object** 各是什么东西之后，我们就能针对它的特性来设计代码了。

1. 原理介绍

其实在笔者刚介绍 Auto Object 特性的时候, 很多读者应该就能猜到它基于什么原理了。在 3.6.3 节中提到过对象的访问器(Accessor)与拦截器(Interceptor)的特性, 也写过两个实例——取自 CNodejs 社区的文章。

所以 Auto Object 的原理其实就是给对象设置一个拦截器, 并且该拦截器内部做的逻辑就是访问当前对象的 `$$access` 函数。

关于拦截器的详细内容, 大家可以翻到 3.6.3 节复习拦截器的相关内容。

2. 目录结构

Auto Object 的目录结构非常简单, 如图 8-1 所示。

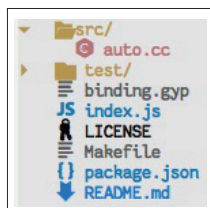


图 8-1 Auto Object 的目录结构

- `src/auto.cc`: C++ 源文件。
- `index.js`: Auto Object 的 JavaScript 封装源码。
- `binding.gyp`: GYP 文件。
- `package.json`: 包定义文件。
- 其他文件。

3. binding.gyp

经过了前面那么多轮的征战, 我们现在闭着眼睛也能想象得到 Auto Object 的 **binding.gyp** 是什么样了。

```
{
  "targets": [{
    "target_name": "auto",
    "sources": [
      "src/auto.cc"
    ],
    "include_dirs": [
      "<!(node -e \"require('nan')\")"
```



```

    ]
  }}
}

```

4. 拦截器函数

Auto Object 的拦截器对象很简单，正如前面所说的那样，给对象设置一个拦截器，并且该拦截器内部做的逻辑就是访问当前对象的 `$$access` 函数。

所以，笔者必须实现定义一个拦截器函数，即 `GenericNamedPropertyGetterCallback`¹。

```

std::string _internal_properties[INTERNAL_PROPERTIES_COUNT] = {
    "constructor",
    ...
    "$$constructor"
};

void PropertyGetter(
    Local<Name> property,
    const PropertyCallbackInfo<v8::Value>& info)
{
    if(!property->IsString())
    {
        return;
    }

    // 忽略被保留的字段 `$$access`
    String::Utf8Value key(property);
    if(!*key || strcmp(*key, "$$access") == 0)
    {
        return;
    }

    // 忽略内置字段
    for(auto i = 0; i < INTERNAL_PROPERTIES_COUNT; i++)
    {
        if(*key == _internal_properties[i])
        {
            return;
        }
    }

    Local<Value> val = Nan::Get(

```

¹ 出自 3.6.3 节，是一个拦截器 `getter` 函数的类型。

```

        info.This(),
        Nan::New<String>("$access").ToLocalChecked()).ToLocalChecked();

    if(val.IsEmpty() || !val->IsFunction())
    {
        Local<Function> emit = Nan::New(emit_warning);
        Local<Value> argv[] = {
            Nan::New("No $access implemented in the object.")->
        ToLocalChecked()
        };
        Nan::Call(emit, info.Holder(), 1, argv);
        return;
    }

    Local<Function> func = val.As<Function>();
    Local<Value> argv[] = { property };

    Nan::TryCatch trycatch;
    MaybeLocal<Value> ret = Nan::Call(func, info.Holder(), 1, argv);
    if(trycatch.HasCaught()) {
        trycatch.ReThrow();
        return;
    }

    info.GetReturnValue().Set(ret.ToLocalChecked());
    return;
}

```

下面一步步解析代码。首先 `_internal_properties` 代表了一些 JavaScript 的内置属性，这些属性将不会被拦截器拦截。

其次是拦截器 `getter` 自身。第一步先是一个判断，看看访问属性的字段是不是一个字符串，若不是字符串则不拦截，取而代之的是去执行普通对象的逻辑，因为在 ECMAScript 6 及更高版本中我们还能通过 `Symbol`¹ 来访问对象的属性。接下来取到访问属性的字段的字符串值，判断是否是 `"$access"` 或者任意一个不需要被拦截的内置属性（即在 `_internal_properties` 列出的属性）。

若这些判断都通过了，我们取到当前对象的 `$access`，看看它是否是一个合法的函数。若不是，则生成一个 `"No $access implemented in the object."` 的错误，并传给 `process.emitWarning()`²。

1 <http://ecma-international.org/ecma-262/#sec-symbol-value>

2 https://nodejs.org/docs/v6.9.4/api/process.html#process_process_emitwarning_warning_name_ctor

在 `Auto Object` 中，`emit_warning` 就是 `process.emitWarning()` 函数，它在该模块初始化的时候被传入并升格成一个持久句柄。

若 `$$access` 是一个合法的函数，则我们就调用 `$$access` 函数并得到结果。在此过程中，为了 `Node.js` 不在 `C++` 层面崩溃，笔者使用了 `Nan::TryCatch` 来捕获 `$$access` 中可能抛出的错误。若的确有错误抛出，这个函数会把错误原封不动地再往外抛出。如果并没有异常被捕获，那么我们就返回 `$$access()` 函数返回的内容。

5. 创建有拦截特性的对象

有了拦截器的函数之后，笔者就能实现创建有拦截特性的对象了。无非就是先创建一个对象模板，然后给这个模板设置拦截器。其中拦截器的 `getter` 是笔者刚写好的 `PropertyGetter`。最后一个参数设置为 `PropertyHandlerFlags::kNonMasking`，即无遮罩模式。对象上面的常规属性拥有更高的优先级。在设置好对象模板之后，通过 `tpl->NewInstance()` 来实例化出来一个对象就可以了。

```
NAN_METHOD(CreateObject)
{
    Local<ObjectTemplate> tpl = Nan::New<ObjectTemplate>();
    tpl->SetHandler(NamedPropertyHandlerConfiguration(
        PropertyGetter,
        0,
        0,
        0,
        0,
        Local<Value>(),
        PropertyHandlerFlags::kNonMasking));
    info.GetReturnValue().Set(tpl->NewInstance());
}
```

`Auto Object` 不支持版本太低的 `Node.js`，因为旧版本的 `V8` 并没有 `NamedPropertyHandlerConfiguration`。这里没有用另一种方式来设置拦截器，是因为笔者需要拦截器配置的最后一个参数 `PropertyHandlerFlags::kNonMasking`。

6. 创建有拦截特性的类

创建一个有拦截特性的类会稍微复杂点儿，毕竟实际上返回的是一个可操作原型链的构造函数。

首先我们要定义类的构造函数，这个函数里面做的事情就是调用原型链上的 `$$constructor` 函数 `__Constructor`。

```

NAN_METHOD(__Constructor)
{
    if(!info.IsConstructCall())
    {
        Nan::ThrowError("Please use `new` to create object.");
        return;
    }

    Nan::TryCatch trycatch;
    Local<Value> val = Nan::Get(
        info.This(),
        Nan::New<String>("$$constructor").ToLocalChecked()).
    ToLocalChecked();

    if(trycatch.HasCaught())
    {
        trycatch.ReThrow();
        return;
    }

    if(val.IsEmpty() || val->IsUndefined() || val->IsNull())
    {
        info.GetReturnValue().Set(info.This());
        return;
    }
    else
    if(!val->IsFunction())
    {
        Nan::ThrowError("Broken object missing `$$constructor`.");
        trycatch.ReThrow();
        return;
    }

    Local<Function> func = val.As<Function>();
    Local<Value> argv[info.Length()];
    for(auto i = 0; i < info.Length(); i++)
    {
        argv[i] = info[i];
    }

    Nan::Call(func, info.This(), info.Length(), argv);
    if(trycatch.HasCaught()) {
        trycatch.ReThrow();
        return;
    }

    info.GetReturnValue().Set(info.This());
}

```

在这个构造函数中，首先用 `info.IsConstructCall()` 判断是否为被通过 `new` 进行访问的，若不是，则抛出一个错误，要求开发者通过 `new` 来实例化对象。

然后就是获取 `$$constructor` 函数并调用了。若 `$$constructor` 不是一个合法的函数，则抛出错误。若在调用 `$$constructor` 过程中有异常，则我们也把异常如实抛出。最后将 `info.This()` 返回即可。

有了构造函数之后，我们就能用构造函数 `__Constructor` 创建一个函数模板，并得到它的 `InstanceTemplate`。

这个 `InstanceTemplate` 就是当构造函数实例化之后的对象对应的对象模板。我们在这个 `InstanceTemplate` 上挂载上自己的拦截器，就能达到实例化出一个对象之后自带拦截效果了。

之后的事情就非常简单了，无非是在 `InstanceTemplate` 挂好拦截器之后，将函数模板实例化成一个函数作为构造函数返回就可以了。

```
NAN_METHOD(CreateClass)
{
    Local<FunctionTemplate> tpl = Nan::New<FunctionTemplate>(__Constructor);
    Local<ObjectTemplate> instance_t = tpl->InstanceTemplate();

    instance_t->SetHandler(NamedPropertyHandlerConfiguration(
        PropertyGetter,
        0,
        0,
        0,
        0,
        Local<Value>(),
        PropertyHandlerFlags::kNonMasking));

    Nan::TryCatch trycatch;
    Local<String> class_name = Nan::New("AutoClass").ToLocalChecked();
    if(info.Length() > 0)
    {
        MaybeLocal<String> maybe = info[0]->ToString();
        if(!maybe.IsEmpty())
        {
            class_name = maybe.ToLocalChecked();
        }
        else
            if(trycatch.HasCaught())
            {
                trycatch.ReThrow();
                return;
            }
    }
}
```

```

    }
}

tpl->SetClassName(class_name);
info.GetReturnValue().Set(tpl->GetFunction());
}

```

在 `CreateClass` 中就是笔者刚才说的过程了，先把 `__Constructor` 封装成一个函数模板，并且通过 `tpl->InstanceTemplate()` 得到它的实例模板。之后就是给模板挂上拦截器（`instance_t->SetHandler()`）。接下去就是读取传入函数的第一个参数，转换为字符串，通过 `tpl->SetClassName()` 来设定将要返回的类的类名。不过中间多了几步判断，以确保类名这个参数的合法性。

最后，返回生成的构造函数，即 `tpl->GetFunction()`。

7. 环境初始化函数

这个函数并不对开发者暴露，而是对笔者的 JavaScript 封装文件暴露，主要就是将 `process.emitWarning` 函数传入并持久化。

```

NAN_METHOD(InitEnv)
{
    Local<Object> param = info[0]->ToObject();
    Local<Function> func = Nan::Get(
        param,
        Nan::New("emitWarning").ToLocalChecked()).ToLocalChecked().
    As<Function>();

    emit_warning.Reset(func);
}

```

对于传进来的参数，我们就自动地认为它就是 `process.emitWarning()` 了，直接把它通过 `emit_warning.Reset()` 进行持久化。毕竟笔者假定这个函数只有我们自己会调用，并不暴露给开发者。

8. 模块初始化函数

把所有的函数都实现好之后，就要进行模块的初始化了。

```

NAN_MODULE_INIT(Init)
{
    Nan::HandleScope scope;

```

```

Nan::Export(target, "createClass", CreateClass);
Nan::Export(target, "createObject", CreateObject);
Nan::Export(target, "init", InitEnv);

Local<Array> array = Nan::New<Array>(INTERNAL_PROPERTIES_COUNT);
for(auto i = 0; i < INTERNAL_PROPERTIES_COUNT; i++)
{
    Nan::Set(array, i, Nan::New<String>(
        _internal_properties[i].c_str()).ToLocalChecked());
}
Nan::Set(target, Nan::New("internalProperties").ToLocalChecked(),
array);
}

```

在函数中，笔者给 `exports` 挂载了 `createClass`、`createObject` 和 `init` 三个函数，以及把 `_internal_properties` 里面的字符串包装成 V8 的数组，暴露到 `exports.internalProperties` 下面。

9. JavaScript 封装

在阅读了前面的一些实例后，大家应该都了解了，一般一个带有 C++ 原生扩展的包，实际暴露出来的是经由 JavaScript 再封装后的接口。所以 Auto Object 也不例外，也经由 `index.js` 进行了一次包装。

首先是一旦执行了 `require("auto-object")`，我们的 `index.js` 模块就会初始化环境，把 `process.emitWarning` 传进去。但是旧版本的 Node.js 并没有这个函数，所以我们有时候需要自己写一个。

```

const auto = require("../build/Release/auto");

function rawEmitWarning(msg) {
    process.emitWarning(msg, "AUTO001", "NoAccessFunction");
}

function fakeEmitWarning(msg) {
    console.warn(`(node) [AUTO001] NoAccessFunction: ${msg}`);
}

auto.init({
    emitWarning: typeof process.emitWarning === "function" ?
        rawEmitWarning :
        fakeEmitWarning
});

```

其次就是 `createClass` 函数了，这个函数其实并不需要任何的封装，所以直接赋值就可以了。同样只需要赋值的是 `internalProperties` 这个内置字段的数组。

```
exports.createClass = auto.createClass;
exports.internalProperties = auto.internalProperties;
```

最后我们再封装一下 `createObject` 函数就可以了。在函数内部，笔者先通过 `auto.createObject()` 创建一个有拦截特性的对象。然后如果参数 `access` 的确是个函数的话，就给对象加上一个 `$$access` 的属性，并且是通过 `Object.defineProperty()` 的形式加上去的，使其变成不可配置的以及枚举不可见的。最后返回该对象。

```
exports.createObject = function(access) {
  const ret = auto.createObject();
  if(typeof access === "function") {
    Object.defineProperty(ret, "$$access", {
      value: access,
      enumerable: false,
      configurable: false,
      writable: true
    });
  }
  return ret;
};
```

8.2.4 小结

本节介绍了一个类似于 Proxy 的包 Auto Object 的源码，顺便温习了笔者之前讲的一些知识点：

- 函数模板；
- 对象模板以及拦截器；
- 持久化句；
- `Nan::TryCatch` 或者 `v8::TryCatch`。

8.2.5 参考资料

[1] 小问. 实战 ES2015：深入现代 JavaScript 应用开发 [M]. 北京：电子工业出版社，2016. 117-127.

[2] 阮一峰. ECMAScript 6 入门 [M]. 北京：电子工业出版社，2014.

9

N-API——下一代 Node.js C++ 扩展开发方式

本章内容在书中将会一带而过，因为在笔者写书的时候，N-API 还没有完全稳定下来，随时会改变。而且笔者个人认为，距离 N-API 能正式投入生产用途的时间还很长。所以本章内容在本书中仅以扩展阅读的形式存在，其实关于 N-API 的内容在 5.1.2 节中曾略微提及。

当你们看到本章中的内容与官方文档不符时，有可能是本书中使用的版本已经落伍了。请以官方文档为准。

就笔者个人而言，对 N-API 持有以下意见。

首先，虽然 N-API 能够事先预编译好二进制的链接库并暴露出接口，看似非常方便，但是其实际上却与系统有关。一旦系统不同，同一个编译好的链接库也是无法适用的。我们很多人的开发机也许是 Macbook，也许是 Windows 系统，而真正在线上运行代码的服务器则通常安装了 Linux 系统。

其次，用纯 C 写扩展，代码的抽象并不会这么好写。当然很多开发者可能更喜欢面向过程的 C 语言，对此笔者也不反对。但是就笔者个人而言，还是更喜欢面向对象的 C++。

再次，就是一旦 N-API 出来之后，对于 Node.js 的 C++ 扩展开发者来说，就不需要去了解内部的更多机制了，可能更方便了。但是，这会减少大家学习 Node.js 底层的动力，以及学习 V8 相关知识的动力。

最后，笔者其实正在观望 WebAssembly¹。

9.1 浅尝辄止

笔者在前文提到过，N-API 自 Node.js v8.0.0 发布之后开始出现，在 Node.js C++ 原生扩展开发的时候，要引入一个 `node_api.h` 的头文件，即：

```
#include <node_api.h>
```

这个头文件中提供了一系列的 N-API 接口和数据类型，而关于 Node.js 的一些函数调用，则变成了：

- 任何 N-API 调用都返回一个 `napi_status` 枚举，以表示这次调用成功与否；
- N-API 的返回值由于被 `napi_status` 占“坑”了，因此真实的返回值由传入的参数来继承，如传入一个指针让函数操作；
- 所有 JavaScript 数据类型都被黑盒类型 `napi_value` 封装，不再是类似于 `v8::Object`、`v8::Number` 等类型；
- 如果函数调用不成功，则可以通过 `napi_get_last_error_info` 函数来获取最后一次出错的信息。

9.1.1 实现一个 Echo 函数

尽管是浅尝辄止，但笔者仍旧会提供一个最简单的样例作为一个基础的参考，让大家对 N-API 的开发有一个最初步的印象。

Echo 函数就是一个你输入什么，就返回你输入的值的函数。

1. 使用 NAN 进行对比

相信对于通读本书的你来说，用 NAN 来实现一个 Node.js 的原生 C++ 扩展应该是小菜一碟。

首先是实现 Echo 函数本身：

¹ 使用 WebAssembly，我们可以在浏览器中运行一些高性能、低级别的编程语言，可用它将大型的 C 和 C++ 代码库比如游戏、物理引擎甚至是桌面应用程序导入 Web 平台，包括 Node.js (<http://webassembly.org/>)。

```
NAN_METHOD(Echo)
{
    if(info.Length() < 1)
    {
        Nan::ThrowError("Wrong number of arguments.");
        return info.GetReturnValue().Set(Nan::Undefined());
    }

    info.GetReturnValue().Set(info[0]);
}
```

然后是模块初始化函数：

```
NAN_MODULE_INIT(InitAll)
{
    Nan::Set(
        target,
        Nan::New<String>("echo").ToLocalChecked(),
        Nan::GetFunction(Nan::New<v8::FunctionTemplate>(Echo)).
        ToLocalChecked());
}
```

一个 C++ 扩展写的 Echo 函数就这么“新鲜出炉”了。

2. 使用 N-API 实现

大家可以打开随书代码的“36. napi/addon.cc”进行阅读。

首先，用 N-API 来写原生扩展，第一个不同就是模块定义的宏不一样了，原来是

```
NODE_MODULE(addon, InitAll)
```

现在变成了

```
// 为了符合纯 C 的风格，在这里把函数名变成了小写
NAPI_MODULE(addon, init)
```

其次，我们如果不通过 NAN 来写的话，InitAll 这个函数的声明应该是这样的：

```
void InitAll(Local<Object> exports)
{
}
```

而使用 N-API 的话，就变成了

```
// 为了符合纯 C 的风格，在这里把函数名变成了小写
void init(napi_env env, napi_value exports, napi_value module, void* priv)
{
}
```

其中，`napi_env env` 用于表示底层 N-API 的特定状态上下文，需要被传入后续嵌套 N-API 调用的函数中作为传递。而 `napi_value exports` 就相当于 `exports` 对象，`module` 是 `module` 对象。

接下来写一个 `echo` 函数。

```
napi_value echo(napi_env env, napi_callback_info info)
{
    napi_status status;

    size_t argc = 1;
    napi_value argv[1];
    status = napi_get_cb_info(env, info, &argc, argv, 0, 0);
    if(status != napi_ok || argc < 1)
    {
        napi_throw_type_error(env, "EBADARGS", "Wrong number of arguments");
        return 0;
    }

    return argv[0];
}
```

注意：在 Node.js v8.0.0 的时候，`napi_throw_type_error()` 还只接收两个参数，分别是 `env` 和 `message`。自 Node.js v8.3.0 之后，该函数添加了第二个参数 `code`，表示错误码¹。这就是一个非常典型的“目前 N-API 还非常不稳定”的例子。因为笔者之前在 GitChat² 的一次直播中讲解 N-API 相关内容³时，样例代码的 `napi_throw_type_error()` 在当前已经无法通过编译了，而那次直播距笔者写本章也就过了一个多月。所以，大家在阅读本书的时候，N-API 可能又已经出现了各种不兼容的更新了。

`napi_value` 函数名 (`napi_env env, napi_callback_info info`) 就是一个 N-API 函数声明的样式了。其中 `env` 就是先前说的用于表示底层上下文的参数，需要逐级传递；而

1 该 Pull Request 的地址是 <https://github.com/nodejs/node/pull/13988>，其修复了 <https://github.com/nodejs/node/issues/13933> 所提到的问题。

2 <http://gitbook.cn/>

3 <http://gitbook.cn/m/mazi/article/593763494ec5fa29296acea0>

info 这个参数的作用类似于传统方式开发 Node.js 的 C++ 扩展时的唯一参数，代表这次调用的一些基本信息，只不过信息的获取方式不同。

当我们需要获取这次调用传进来的 JavaScript 参数时，是通过 `napi_get_cb_info()` 函数来做的，传入 `env` 和 `info`，并传入接收参数的两个变量 `argc` 和 `argv`。`status` 就是获取参数的结果，在前文也提到过，任何 N-API 调用都返回一个 `napi_status` 枚举，以表示这次调用成功与否。如果参数获取失败，则通过 `napi_throw_type_error()` 抛出一个错误，并返回 0（`napi_value` 实际上是一个指针，返回空指针表示无返回值）。最后返回 `argv` 第一位即可，也就是传入参数的第一个参数。

这就是一个最简单的 `echo` 函数的实现了。接下来开始补全 `init` 函数。

```
void init(napi_env env, napi_value exports, napi_value module, void* priv)
{
    napi_status status;

    // 用于设置 exports 对象的描述结构体
    napi_property_descriptor desc =
        { "echo", 0, echo, 0, 0, 0, napi_default, 0 };

    // 把 "echo" 设置到 exports 中
    status = napi_define_properties(env, exports, 1, &desc);
}
```

首先还是设置一个 `status` 变量，然后声明一个用于设置 `exports` 对象的描述结构体 `napi_property_descriptor desc`。

`napi_property_descriptor` 是用于设置对象属性的描述结构体，它的声明如下：

```
typedef struct {
    const char* utf8name;
    napi_value name;

    napi_callback method;
    napi_callback getter;
    napi_callback setter;
    napi_value value;

    napi_property_attributes attributes;
    void* data;
} napi_property_descriptor;
```

在 `desc` 中，它所描述的意思如下：目标对象下会挂一个叫 `"echo"` 的东西，它对应的内容是函数 `echo`，其他的 `getter` 和 `setter` 等全是空指针，而它的属性则是 `napi_default`。

`napi_property_attributes` 除了 `napi_default` 之外, 还有诸如只读、是否可枚举等属性。

9.1.2 尝试运行 N-API 扩展

1. 命令行执行

首先在 Node.js v8.3.0 下进行试验吧, 执行下面的命令:

```
$ node-gyp rebuild
...
$ node --napi-modules
(node:31084) Warning: N-API is an experimental feature and could change at
any time.
> const addon = require("./build/Release/addon");
undefined
> addon.echo("2333");
'2333'
> addon.echo("蛋花汤 🍲", "南瓜饼 🍪");
'蛋花汤 🍲'
> addon.echo();
TypeError [EBADARGS]: Wrong number of arguments
    at repl:1:7
    at ContextifyScript.Script.runInThisContext (vm.js:44:33)
    at REPLServer.defaultEval (repl.js:239:29)
    at bound (domain.js:301:14)
    at REPLServer.runBound [as eval] (domain.js:314:12)
    at REPLServer.onLine (repl.js:433:10)
    at emitOne (events.js:120:20)
    at REPLServer.emit (events.js:210:7)
    at REPLServer.Interface._onLine (readline.js:278:10)
    at REPLServer.Interface._line (readline.js:625:8)
```

注意: 还是因为试验特性, 目前在 Node.js v8.x 要加载和执行 N-API 的 C++ 扩展的话, 在启动 node 时需要加上 `-napi-modules` 参数, 表示这次执行要启用 N-API 特性。

效果显而易见, 在刚启动 Node.js REPL 的时候, 你会得到一个警告。

```
(node:52264) Warning: N-API is an experimental feature and could
change at any time
```

表示它目前还不是特别稳定, 但是未来值得期待。然后在通过 `require()` 载入扩展的时候, 我们就得到了一个拥有 `echo` 函数的对象了。

笔者尝试了 3 种调用方式。第一次是规规矩矩传入一个参数，echo 如期返回传入的参数 "2333"; 第二次传入两个参数，echo 返回了第一个参数 "蛋花汤 🍲"; 最后一次没传任何参数，这个时候就走到了 C++ 扩展中判断函数参数数量失败的条件分支，于是抛出了一个 `Wrong number of arguments` 的错误对象。

总之，它按照我们的预期跑起来了，并且代码里面并没有任何 Node.js 非 N-API 所暴露出来的数据结构和 V8 的数据结构——版本差异消除了。

2. 换个版本执行

接下来激动人心的时刻到了。如果读者使用 `nvm` 来管理自己的 Node.js 版本，则可以尝试着安装一个 8.4.0 及以上兼容当前代码的 Node.js 版本。

```
$ nvm install 8.3.0
```

在安装成功、切换版本成功后，尝试着直接打开 Node.js REPL，忘掉“由于 Node.js 版本不同所导致的 C++ 扩展需要重新编译”这一步。不过执行的时候别忘了加上 `--napi-module` 参数。

把刚才用于测试的几句 JavaScript 代码再重复地输入——N-API 诚不我欺，居然仍能输出结果。这对于以前的暴力做法和 NAN 做法来说，无疑是非常大的一个进步。

9.1.3 向下兼容

至此，笔者希望大家还没有忘记 N-API 是自 Node.js 8.0 之后出的特性。所以，之前 Demo 的代码并不能在 Node.js 8.0 之前的版本中如期编译和运行。

1. node-addon-api

辛辛苦苦写好的包，居然不能在 Node.js 6.x 下执行，这怎么可以呢，此刻读者的心情是不是如图 9-1 中这样？



图 9-1 心情

先别着急。其实还有一个外挂式头文件的包，其包名是 `node-addon-api`。

我们就试着通过它来进行向下兼容吧。首先在刚才的源码目录中安装这个包。

```
$ npm install --save node-addon-api
```

还是由于快速迭代的原因，笔者不能保证这个包在当前版本的时效性。不过笔者相信大家都有探索精神，在未来新版本的 Node.js 中由于版本不符而导致的 API 不兼容问题，应该都能解决。

2. binding.gyp 的修改

然后，给我们的 `binding.gyp` 函数加点“料”——加两个字段，里面是两个指令展开。

```
"include_dirs": [ "<!(node -p \"require('node-addon-api').include\")" ],
"dependencies": [ "<!(node -p \"require('node-addon-api').gyp\")" ]
```

`<!@` 和 `<!` 开头的字符串在 GYP 中代表指令，表示它的值是后面指令的执行结果，这在 4.1.3 节中有详细说明。上面两条指令的返回结果分别是外挂式头文件的头文件搜索路径，以及外挂式 N-API 这个包编译成静态链接库供我们自己的包使用的依赖声明。

有了这两个字段后，就表示我们依赖了外挂式 N-API 头文件。而且它内部自带判断。该判断会在 Node.js 的版本已经达到了“有 N-API”的要求的时候，将该依赖留空。即不依赖外挂式 N-API 编译的静态链接库。**也就是说，用了外挂式的 N-API，能自动适配 Node.js 8.x 和低版本。**

于是这个 `binding.gyp` 现在看起来是这样的：

```
{
  "targets": [{
    "target_name": "addon",
    "sources": [
      "addon.cpp"
    ],
    "include_dirs": [ "<!(node -p \"require('node-addon-api').include\")" ],
    "dependencies": [ "<!(node -p \"require('node-addon-api').gyp\")" ]
  }]
}
```

注意：在当前版本的 `node-addon-api` 中，建议在你的代码项目目录下不要出现空格以及其他一些非常规类的目录字符，否则会出现编译失败的问题。之后的版本也许会修复该问题，大家也可以在新版本中试试看该问题是否还存在。

至于源码层面，我们就不需要做任何修改了。在 Node.js v6.x 下试试看吧。同样也是使用 `node-gyp rebuild` 进行编译。然后通过 Node.js REPL 进行测试。

具体的终端输出这里就不放出来了，相信经过试验的大家都得到了自己想要的结果。（这回在低版本中就不需要加 `--napi-modules` 参数了。）

9.1.4 N-API Package——C++ 封装

N-API Package 是一个基于 N-API 的 C++ 封装的头文件包，它对 N-API 进行了封装，使开发者能用 C++ 的形式（面向对象等）进行开发，且底层调用的是 N-API。

N-API Package 的仓库地址是 <https://github.com/nodejs/node-addon-api>，由于篇幅和精力的原因，本书就不对其进行详解了，有兴趣的读者可以自行前往其源码仓库进行进一步的了解。

9.1.5 小结

本节初步讲解了初期 N-API 的一些使用姿势，虽然未来的走势不是特别稳定，但也可以给大家展示出 N-API 的一些基本思想，以及一个初步的印象。

9.1.6 参考资料

[1] 从暴力到 NAN 再到 NAPI——Node.js 原生模块开发方式变迁: <https://xcoder.in/2017/07/01/nodejs-addon-history/>.

9.2 基本数据类型与错误处理

本节将介绍 N-API 的基本数据类型，不过书面上的内容会与实际的最新内容有出入，请以官方文档为准。鉴于此，本节不会有实战环节，而是讲解略微宽泛的理论。而本节中讲述的 N-API 所对应的 Node.js 都为 v8.4.0。

9.2.1 基本数据类型

N-API 包含了多种抽象的基本数据类型供大家使用。这些基础类型分别为：

- `napi_status`;
- `napi_extended_error_info`;
- `napi_env`;
- `napi_value`。

1. napi_status

这个数据类型的变量用于接收 N-API 各函数调用的状态结果——即调用是否成功。在 9.1 节的样例中就有体现，如：

```
napi_status status = napi_define_properties(env, exports, 1, &desc);
```

上面代码中的 `status` 表示给 `exports` 对象挂上 `desc` 中所描述的属性这个操作的成功状态——若为 `napi_ok`，就表示成功。

实际上，`napi_status` 是一个枚举。其目前的声明如下：

```
typedef enum {  
  napi_ok,  
  napi_invalid_arg,  
  napi_object_expected,  
  napi_string_expected,  
  napi_name_expected,  
  napi_function_expected,  
  napi_number_expected,  
  napi_boolean_expected,  
  napi_array_expected,  
  napi_generic_failure,  
  napi_pending_exception,  
  napi_cancelled,  
  napi_status_last  
} napi_status;
```

当你调用了一个 N-API 的函数并得到了一个 `napi_status` 结果的时候，如果它不是成功状态的话，我们就能通过调用 `napi_get_last_error_info()` 函数来获取最后一次调用的失败详细信息。

2. napi_extended_error_info

这个数据类型就是调用失败的详细信息了，具体可以参照后文错误处理的内容。`napi_extended_error_info` 实际上是一个结构体，声明如下：

```
typedef struct {  
  const char* error_message;  
  void* engine_reserved;  
  uint32_t engine_error_code;  
  napi_status error_code;  
} napi_extended_error_info;
```

结构体中的 4 个属性分别代表：

- `error_message`——UTF-8 编码的错误描述信息。
- `engine_reserved`——引擎预留字段，当前还没用。
- `engine_error_code`——引擎错误码，当前还没实现。
- `error_code`——错误码。

3. `napi_env`

这个数据类型在前面曾提到过，用于表示底层 N-API 的特定状态上下文，需要被传入后续嵌套 N-API 调用的函数中作为传递。我们不需要关心它内部到底是什么东西、有什么东西，只管按照 N-API 的各种函数声明，在调用的时候老实地传递下去就可以了。

4. `napi_value`

这是一个指针，声明如下：

```
typedef struct napi_value__ *napi_value;
```

但是，我们也不需要知道 `napi_value__` 内部具体有些什么东西。在 N-API 的基本数据类型里面，这些内容通常只是给 N-API 的内部进行解析用的，它只是一个抽象的数据类型。如果我们需要得到一些详细的数据如 `int`、`double` 等，只需要将 `napi_value` 传给一些 N-API 解析函数进行解析就可以了。比如，我们如果要将一个 `napi_value` 所代表的布尔值取出来，只需要调用这样一个函数：

```
napi_status napi_get_value_bool(napi_env env, napi_value value, bool* result)
```

关于 `napi_value` 的各种数据操作，本书就不再赘述了。由于目前暴露的 API 还不是特别稳定，还请大家以官方文档（https://nodejs.org/docs/v8.4.0/api/n-api.html#n-api_working_with_javascript_values）为准进行阅读。

9.2.2 与作用域及生命周期相关的数据类型

在 V8 中，有句柄作用域的概念，它是通过 `HandleScope` 这个类型来实现的。而在 N-API 中，同样有供大家实现类似概念的数据类型。

通常，N-API 的一些变量本身就是在在一个上下文的句柄作用域中被创建出来的——与 Node.js 传统模式下的 C++ 扩展开发类似，在一个 N-API 的原生扩展中函数被调用时，Node.js

已经为你创建好了外层的句柄作用域。也就是说，在这些调用函数中如果我们不自己去生成一个句柄作用域的话，在这些函数内部生成的所有变量都会沿用先前 Node.js 创建好的句柄作用域的生命周期——也就是说，通常情况下一个 `napi_value` 的生命周期均小于它所处的函数本身。

但是有时候，我们的函数不一定在 Node.js 主事件循环的辖控之中（如在 libuv 回调的调用期间），这个时候我们可能需要自己创建一个句柄作用域。

1. `napi_handle_scope`

这是一个用来控制以及修改在作用域中创建出来的对象的生命周期的数据类型。与 V8 原生的句柄作用域数据类型不同的是，当我们声明了一个 `napi_handle_scope` 的变量时，还需要自行打开这个作用域，并且在必要的时候结束作用域。

- `napi_status napi_open_handle_scope(napi_env, napi_handle_scope*)`：打开一个作用域。
- `napi_status napi_close_handle_scope(napi_env, napi_handle_scope*)`：关闭一个作用域。

2. `napi_escapable_handle_scope`

这个数据类型类似于 V8 的可逃句柄作用域。

它的开启作用域和关闭作用域的函数名不一样，但参数可以等同。两个函数名分别是 `napi_open_escapable_handle_scope` 和 `napi_close_escapable_handle_scope`。

在作用域的开启期间，我们能通过 `napi_escape_handle` 函数指定逃脱作用域的数据：

```
napi_status napi_escape_handle(napi_env, napi_escapable_handle_scope, napi_value escapee, napi_value* result)
```

其中，第一个参数就是我们每次都要传的 `napi_env`；第二个参数就是句柄作用域本体，`escapee` 代表我们待逃脱的 N-API 数据类型，也就是包裹着 V8 实体的一个待逃者；而最后一个参数 `result` 是一个用于接收逃脱后的 `napi_value` 数据的指针——也就是说当 `escapee` 逃脱后，它会变成一个新的 `napi_value`，然后赋值到 `result` 还给你，我们之后在作用域之外能用的就是这个 `result` 了。

3. napi_ref

这是 N-API 中另一个用于管理对象生命周期的数据类型，可以理解为引用计数。我们能为一个 `napi_value` 对象创建一个引用计数变量即 `napi_ref`，并通过为其计数加一或者减一来操作其生命周期。

在某些特定的场景中，我们可能需要创建一个比“只存在于单个函数中的 `napi_value` 对象”的生命周期更为长久的对象。例如，当我们创建一个构造函数，然后将在之后的一些函数调用中通过这个构造函数来生成对象的时候，这个构造函数应该被引用计数所操纵，使其不是马上就失效——因为正如前文所说，通常情况下一个 `napi_value` 的生命周期均小于它所处的函数本身，这个构造函数会在生成它的函数结束之前被销毁。

所以其做法就有点类似于 V8 持久句柄了，只不过对于开发者来说其用法不太一样。

要创建引用计数变量，是通过 `napi_create_reference()` 函数进行的：

```
napi_status napi_create_reference(napi_env env,
                                napi_value value,
                                int initial_refcount,
                                napi_ref* result);
```

第二个参数就是要与待创建的引用计数绑定在一起的 `napi_value` 变量，第三个参数是引用计数的初始计数值，最后一个参数用于接收生成好的引用计数变量。

当我们要给引用计数加一或者减一的时候，是通过 `napi_reference_ref` 或者 `napi_reference_unref` 函数来完成的：

```
napi_status napi_reference_ref(napi_env env,
                              napi_ref ref,
                              int* result);

napi_status napi_reference_unref(napi_env env,
                                napi_ref ref,
                                int* result);
```

这两个函数的第二个参数和第三个参数分别代表引用计数的变量本体以及用于告知开发者新的引用计数值的一个 `int*` 指针。

当我们需要知道这个引用计数所对应的 `napi_value` 是什么的时候，是通过 `napi_get_reference_value` 函数来完成的：

```
napi_status napi_get_reference_value(napi_env env,
                                     napi_ref ref,
                                     napi_value* result);
```

其参数非常简单，一目了然。

其他有关 `napi_ref` 的函数，大家可以详细阅读 N-API 的官方文档来获取更多信息。

9.2.3 回调数据类型

N-API 有以下几种回调相关类型，供开发者使用。

- `napi_callback_info`;
- `napi_callback`;
- `napi_finalize`;
- `napi_async_execute_callback`;
- `napi_async_complete_callback`。

本节将讲述 `napi_callback_info` 和 `napi_callback`，对于其他 3 个数据类型，大家可将其作为扩展阅读内容，并可前往 Node.js 官方文档获取其详细信息。

1. `napi_callback_info`

这个数据类型对于开发者来说不透明，只是一个抽象，开发者也不需要知道它里面到底有什么东西。其在 9.1 节的样例中有所体现，它就是传入一个 N-API 的 JavaScript 层调用函数的参数类型。

```
napi_value echo(napi_env env, napi_callback_info info)
{
}
```

要拿到它里面的一些信息，必须借助于一些 N-API 的函数。如在样例中获取这个函数的 JavaScript 层参数，就是这样做的：

```
size_t argc = 1;
napi_value argv[1];
status = napi_get_cb_info(env, info, &argc, argv, 0, 0);
```

2. napi_callback

这个类型实际上是一个 typedef，它指的就是我们声明的能供 JavaScript 调用的函数的骨架，在样例中就是这样一个函数声明：

```
napi_value echo(napi_env env, napi_callback_info info)
{
}
```

napi_callback 的本体声明是这样的：

```
typedef napi_value (*napi_callback)(napi_env, napi_callback_info);
```

9.2.4 错误处理

N-API 的错误处理有两种：一种就是函数调用时的返回值，即通过 napi_status 来辨别；另一种就是 JavaScript 层面的异常。

1. napi_status 返回值

前文曾说过，所有 N-API 的函数返回值都是一个 napi_status。

如果返回值的结果是 napi_ok，就说明调用是成功的。如果有错误发生并且有异常出现，那么返回的则是相应错误的结果。如果异常是“需要的值是字符串”的话，结果就是 napi_string_expected。但如果有异常抛出但是没有错误发生的话，返回值就是一个 napi_pending_exception 了。

当返回值不是 napi_ok 也不是 napi_pending_exception 的时候，我们必须调用一下 napi_is_exception_pending() 函数来确认是否有一个抛出的异常正在等待处理，这在后文会提到。

如果之后 napi_status 的枚举值有所变动，则以 Node.js 源码头文件的 src/node_api_types.h 文件中的定义为准。

当我们拿到一个错误的 napi_status 值时，我们能通过 napi_get_last_error_info() 函数来获取它的详细信息。

```
napi_status
napi_get_last_error_info(napi_env env,
                        const napi_extended_error_info** result);
```

注意：在错误扩展信息中的任何内容都只能作为记录用，我们不能将其作为一些条件判断（如认为当信息中存在某个字符串的时候就是某一种错误），因为这个扩展信息不受 SemVer 的约束，并可能随时修改。

2. 异常

所有的 N-API 函数调用结果都可能是一个待处理的 JavaScript 异常。为了尝试恢复代码的执行，而不是简单粗暴地返回某个结果，我们需要通过调用 `napi_is_exception_pending` 才能判断是否有异常抛出并被挂起。当有异常待处理的时候，采取以下两种方法。

第一种方法是做少量的清理并且直接返回，这样一来代码的执行会返回到 JavaScript 层面。一旦这么做，刚才抛出的异常就会在 JavaScript 层的代码中继续被抛出——也就是说把异常抛给 JavaScript 层去处理。大多数的 N-API 函数在触发异常抛出的时候，行为是不确定的，尤其是很多函数的行为只是简单粗暴地返回 `napi_pending_exception`。所以我们需要做的就是尽可能少地破坏现场（也就是说尽可能不做一些额外的事），并通过直接返回来将异常继续往上层抛。

如果你想在 C++ 代码层面就把异常处理掉的话，就需要用第二种方法了。有些情况下，异常是可以被捕获的，然后就可以采取适当的操作继续执行。只有在已知的可以安全处理异常的特定情况下才推荐使用这种方法。在这些情况下，可以使用 `napi_get_and_clear_last_exception()` 函数来获取和清除异常。当该函数执行成功时，结果就会包含抛出的最后一个异常的 JavaScript 对象的句柄。接下来就可以针对异常做一些事情了。如忽略异常，然后做一些矫正的返回，或者当发现这个异常我们无法在 C++ 层面继续处理的时候，仍然可以选择通过 `napi_throw()` 函数来重新抛出这个异常。

```
napi_status napi_get_and_clear_last_exception(napi_env env,
                                              napi_value* result);

napi_status napi_throw(napi_env env, napi_value error);
```

如果你自己想抛出一个新的异常，而不是把一个获得的异常重新抛出去的话，有 3 个函数可以用：

```
napi_status napi_throw_error(napi_env env,
                             const char* code,
                             const char* msg);

napi_status napi_throw_type_error(napi_env env,
                                  const char* code,
                                  const char* msg);
```



```
napi_status napi_throw_range_error(napi_env env,
                                   const char* code,
                                   const char* msg);
```

其中 `napi_throw_type_error()` 已在浅尝辄止的源码样例中使用过了，其他两个函数的用法也基本一样。

除了抛出异常，N-API 还提供了一个函数用于检测一个 `napi_value` 是否为一个异常对象。

```
napi_status napi_is_error(napi_env env,
                          napi_value value,
                          bool* result);
```

其他更多有关 N-API 异常的函数，大家可以阅读 Node.js 的官方文档来了解详情。

3. 致命错误 (Fatal Error)

在 C++ 扩展中，如果发生了一个不可恢复的错误时，我们可能需要抛出一个致命错误 (Fatal Error) 来马上终止进程。

```
void napi_fatal_error(const char* location, const char* message);
```

`napi_fatal_error()` 函数有别于其他一些 N-API 函数之处是，它没有任何的返回值，一经调用则进程立即终止。

9.2.5 模块注册

N-API 的模块注册与传统的 Node.js C++ 扩展注册类似，但有以下两个不同点。

一是只需要把 `NODE_MODULE` 这个宏用 `NAPI_MODULE` 代替即可。

```
NAPI_MODULE(addon, Init)
```

二是 `Init` 函数的声明变得不一样了，在 N-API 中，`Init` 类似于下面这样：

```
void Init(napi_env env, napi_value exports, napi_value module, void* priv);
```

与传统的 Node.js C++ 扩展开发一样，在这个函数中通过给传入的 `exports` 或者 `module` 对象添加一些属性来达到导出的效果，就像这样：

```
void Init(napi_env env, napi_value exports, napi_value module, void* priv) {
    napi_status status;
    napi_property_descriptor desc =
        {"hello", Method, 0, 0, 0, napi_default, 0};
    status = napi_define_properties(env, exports, 1, &desc);
}
```

上面这段代码会导出一个 `exports.hello` 函数。如果需要替换整个 `module.exports` 的话，可以这么做：

```
void Init(napi_env env, napi_value exports, napi_value module, void* priv) {
    napi_status status;
    napi_property_descriptor desc =
        {"exports", Method, 0, 0, 0, napi_default, 0};
    status = napi_define_properties(env, module, 1, &desc);
}
```

9.2.6 小结

本节介绍了 N-API 的基本数据类型和使用方法。虽然在未来的版本中极有可能会变化，但是基本的理念应该不会变，读者可以选择性地阅读其重要的思想相关部分。

9.2.7 参考资料

[1] N-API | Node.js v8.4.0 Documentation: <https://nodejs.org/docs/v8.4.0/api/n-api.html>.

9.3 对象与函数

与前面的章节一样，本节内容只讲浅显的理论，不做实践。而且作为本章的最后一节内容，后续不会再有更深入的内容，比如 N-API 中的异步等，这些内容的更多信息，大家可以参考官方文档。

9.3.1 对象

本节将会列举 N-API 中对象的一些基本操作。

1. 对象的创建

在 Node.js 中，内置的一些对象有很多种，包括数组、Buffer、ArrayBuffer¹、函数、符号（Symbol）、TypedArray² 等。

创建这些对象，N-API 提供了如下函数。

- `napi_create_array`: 创建一个空数组。
- `napi_create_array_with_length`: 创建一个指定长度的数组。
- `napi_create_arraybuffer`: 创建一个 ArrayBuffer。
- `napi_create_buffer`: 创建一个 Node.js 中的 Buffer 对象。
- `napi_create_buffer_copy`: 用 Copy 模式创建一个 Buffer，与 `Nan::CopyBuffer()`³ 相似。
- `napi_create_external`: 创建一个本体是 `v8::External` 的 JavaScript 元素。
- `napi_create_external_arraybuffer`: 创建一个 ArrayBuffer，并在创建的过程中传入一个 `v8::External` 对象。
- `napi_create_external_buffer`: 创建一个 Buffer，并多了一点参数，如 `napi_finalize`。
- `napi_create_function`: 根据一个 `napi_callback` 类型的 C++ 函数创建相应的 JavaScript 函数。
- `napi_create_object`: 创建一个空对象。
- `napi_create_symbol`: 创建一个 Symbol。
- `napi_create_typedarray`: 创建一个 TypedArray。
- `napi_create_dataview`: 创建一个 DataView⁴。

这些函数的字面意义都比较明显，参数和用法可以详细参考官方文档（https://nodejs.org/docs/v8.4.0/api/n-api.html#n_api_object_creation_functions）。

1 ArrayBuffer 对象用于表示一个固定长度的 Buffer，参考网址：<https://tc39.github.io/ecma262/#sec-arraybuffer-objects>。

2 TypedArray 对象是一个类数组的 Buffer，其中各元素的类型是固定的。大体有 3 种类型的 TypedArray，分别是无符号整数、有符号整数以及浮点数，参考网址：<http://www.ecma-international.org/ecma-262/6.0/#sec-typedarray-objects>。

3 参照 5.4.3 节 Buffer 的相关内容。

4 <https://tc39.github.io/ecma262/#sec-dataview-objects>。

2. 对象属性的基本操作

N-API 提供了一系列的 API 来供开发者对 JavaScript 的对象进行属性的读 / 写操作。

我们知道, JavaScript 的属性实际上就是一个键值对, 从本质上说, N-API 中所有的属性键名都可以是这几种形式之一。

- **命名属性:** 一个 UTF-8 编码的字符串。
- **数值属性:** 一个索引值, 即一个无符号整数 `uint32_t`。
- **JavaScript 属性:** 一个 `napi_value` 类型, 但是它的本体需要是 `v8::String`、`v8::Number` 或者 `v8::Symbol`。

假设我们有这么一段 JavaScript 的代码:

```
const obj = {};
obj.myProp = 123;
```

如何用传统的 Node.js C++ 原生扩展方式来书写我们已经轻车熟路了, 但是如果要用 N-API 来完成一段等价的话, 应该是这样的:

```
napi_status status;

// const obj = {}
napi_value obj, value;
status = napi_create_object(env, &obj);
if (status != napi_ok) return status;

// 创建一个值是 123 的 `napi_value`
status = napi_create_int32(env, 123, &value);
if (status != napi_ok) return status;

// obj.myProp = 123
status = napi_set_named_property(env, obj, "myProp", value);
if (status != napi_ok) return status;
```

看着注释, 这段代码应该不需要过多解释了, 相信大家都能看得懂。类似地, 笔者如果用索引键名 (也就是数组) 的话, 可以写出这样一段 JavaScript 代码:

```
const arr = [];
arr[123] = 'hello';
```

那么, 其等价于:

```
napi_status status;

// const arr = [];
napi_value arr, value;
status = napi_create_array(env, &arr);
if (status != napi_ok) return status;

// 创建一个值是 'hello' 的 `napi_value`
status = napi_create_string_utf8(env, "hello", -1, &value);
if (status != napi_ok) return status;

// arr[123] = 'hello';
status = napi_set_element(env, arr, 123, value);
if (status != napi_ok) return status;
```

当然，有写就有读，我们可以写一段 JavaScript 代码来读取某个属性：

```
const arr = [];
const value = arr[123];
```

用 N-API 来写就是这样的：

```
napi_status status;

// const arr = []
napi_value arr, value;
status = napi_create_array(env, &arr);
if (status != napi_ok) return status;

// const value = arr[123]
status = napi_get_element(env, arr, 123, &value);
if (status != napi_ok) return status;
```

最后，我们再来尝试一下别的内容——模拟一个 `Object.Properties()`：

```
const obj = {};
Object.defineProperties(obj, {
  'foo': { value: 123, writable: true, configurable: true, enumerable: true },
  'bar': { value: 456, writable: true, configurable: true, enumerable: true }
});
```

上面的代码使用 N-API 来实现就是这样的：

```

napi_status status;

// const obj = {};
napi_value obj;
status = napi_create_obj(env, &obj);
if (status != napi_ok) return status;

// 创建值为 123 和 456 的 `napi_value`
napi_value fooValue, barValue;
status = napi_create_int32(env, 123, &fooValue);
if (status != napi_ok) return status;
status = napi_create_int32(env, 456, &barValue);
if (status != napi_ok) return status;

// 设置属性
napi_property_descriptors descriptors[] = {
    { "foo", fooValue, 0, 0, 0, napi_default, 0 },
    { "bar", barValue, 0, 0, 0, napi_default, 0 }
}
status = napi_define_properties(env,
                                obj,
                                sizeof(descriptors) /
sizeof(descriptors[0]),
                                descriptors);
if (status != napi_ok) return status;

```

3. 对象属性设置的数据结构

前文中笔者讲述了对对象属性设置的基本操作，其中在设置属性的时候，代码中出现了 `napi_property_descriptors` 这个数据结构。它其实是一个结构体，与 `napi_property_attributes` 配合使用，可以给一个对象设置属性。

- `napi_property_attributes`: 这是一个枚举，用于表示属性的性质，如读/写性质、是否可被枚举等。它的声明如下：

```

typedef enum {
    napi_default = 0,
    napi_writable = 1 << 0,
    napi_enumerable = 1 << 1,
    napi_configurable = 1 << 2,

    // 与 napi_define_class 一起使用，以区分静态属性和实例属性。同时会被
    // napi_define_properties 忽略。
    napi_static = 1 << 10,
} napi_property_attributes;

```

- 默认状态下（即 `napi_default`）是只读、不可枚举以及不可配置的。要添加哪项性质，只需要在 `napi_default` 上做或操作就可以了。例如可写、可枚举以及可配置的值就是 `napi_writable | napi_enumerable | napi_configurable`。
- `napi_property_descriptor`：这个数据结构在 9.1 节中曾提到过，其声明如下：

```
typedef struct {
    // 其中 `utf8name` 和 `name` 有一项需要为 NULL
    const char* utf8name;
    napi_value name;

    napi_callback method;
    napi_callback getter;
    napi_callback setter;
    napi_value value;

    napi_property_attributes attributes;
    void* data;
} napi_property_descriptor;
```

4. 对象属性相关函数

操作对象属性所相关的数据结构在前文曾提过，但是“有了米之后，还要有巧妇去炊”。对于对象属性的操作，是通过这些函数结合前文提到的数据结构一起来进行的。

- `napi_get_property_names`：获取一个对象的属性键名数组，类似于 `Object.keys()`。
- `napi_set_property`：给一个对象设置一个属性。
- `napi_get_property`：获取一个对象指定的一个属性。
- `napi_has_property`：判断一个对象是否有指定的一个属性。
- `napi_delete_property`：删除一个对象的某个指定的属性。
- `napi_has_own_property`：类似于 `Object.hasOwnProperty()`。
- `napi_set_named_property`：给一个对象设置一个命名属性。
- `napi_get_named_property`：获取一个对象下的一个指定命名属性。
- `napi_has_named_property`：判断一个对象下是否有一个指定的命名属性。
- `napi_set_element`：给一个对象设置一个数值属性。
- `napi_get_element`：获取一个对象下的一个指定数值属性。
- `napi_has_element`：判断一个对象下是否有一个指定的数值属性。

- `napi_delete_element`: 删除一个对象下的一个指定数值属性。
- `napi_define_properties`: 通过 `napi_property_descriptor` 给一个对象设置多个属性。

这些函数的字面意义都比较明显，其参数和用法可以详细参考官方文档（<https://nodejs.org/docs/v8.4.0/api/n-api.html#n-api-functions>）。

9.3.2 函数

1. JavaScript 调用 N-API 函数

N-API 提供了一系列的 API 以让 JavaScript 来调用 C++ 原生的扩展。笔者在 N-API 中实现了一个能直接供 JavaScript 调用的函数，是通过声明一个 `napi_callback` 的函数来做到的，这在 9.2.3 节中也曾提到过。而这个 `napi_callback` 函数中做的操作是这样的：

- 得到一个上下文的信息，即 `napi_env`，供其透传。
- 得到传入该函数的各种参数以及调用信息，即 `napi_callback_info`。
- 在函数中返回一个 `napi_value` 表示其返回结果。

此外，N-API 还提供了一系列方法供 C++ 层面的代码去调用 JavaScript 的函数，常规的函数以及构造函数都可以。

2. N-API 调用 JavaScript 函数——`napi_call_function`

这个函数用于在 N-API 层面的代码中去同步调用一个 JavaScript 的函数。函数原型如下：

```
napi_status napi_call_function(napi_env env,
                               napi_value recv,
                               napi_value func,
                               int argc,
                               const napi_value* argv,
                               napi_value* result)
```

- `env`: 上下文数据。
- `recv`: 相当于 `Function.prototype.call(thisArg, ...)` 中的 `thisArg`。
- `func`: 即将被调用的函数。
- `argc`: 调用该函数时的参数数量。
- `argv`: 调用该函数时的参数数组。
- `result`: 函数执行结果。

假设我们在 JavaScript 的 `global` 对象中有一个 `addTwo` 的函数如下：

```
function addTwo(num) {
    return num + 2;
}
```

那么，就能在 N-API 中通过 `napi_call_function` 去执行它：

```
napi_value global, add_two, arg;

// 获取 `global` 对象，函数详情请参考官方文档
napi_status status = napi_get_global(env, &global);
if (status != napi_ok) return;

// 从 `global` 对象中获取 `addTwo`
status = napi_get_named_property(env, global, "addTwo", &add_two);
if (status != napi_ok) return;

// 等同于 const arg = 1337
status = napi_create_int32(env, 1337, &arg);
if (status != napi_ok) return;

// 等同于创建一个对象数组
// napi_value argv[] = { arg };
napi_value* argv = &arg;
size_t argc = 1;

// 执行 addTwo(arg); 并得到结果
napi_value return_val;
status = napi_call_function(env, global, add_two, argc, argv, &return_val);
if (status != napi_ok) return;

// 将结果转换回 C++ 的 `int32_t` 数据
int32_t result;
status = napi_get_value_int32(env, return_val, &result);
if (status != napi_ok) return;
```

3. 创建一个 N-API 函数——`napi_create_function`

`napi_create_function` 用于创建一个能在 JavaScript 调用的函数，这类似于传统 Node.js C++ 扩展开发的生成一个函数模板并从中实例化出一个函数。与传统方式相似，函数创建好后对外部是不可见的，必须要挂载到某个媒介中导出才有用，如 `module`、`global` 或者传入的参数里等。

函数的声明如下：

```
napi_status napi_create_function(napi_env env,
                                const char* utf8name,
                                napi_callback cb,
                                void* data,
                                napi_value* result);
```

- env: 上下文。
- utf8name: 函数名。
- cb: C++ 写的函数本体，为 `napi_callback` 类型。
- data: 每次调用这个函数的时候，这个 data 会被传入以供使用。
- result: 用于接收通过该函数创建出来的 N-API 函数本体，即可供 JavaScript 调用的 function。

具体的使用样例可参考 9.1 节，大家可以回过头去阅读一下。

4. 获取函数调用信息——`napi_get_cb_info`

在一个我们写的 N-API 的函数中，第一个参数是 `napi_env` 的上下文，第二个参数则是一个 `napi_callback_info` 的代表当前函数调用的一些信息数据，如调用参数等。

在前文中提到过，它对开发者来说不透明，只是一个抽象。那么我们就能通过 `napi_get_cb_info` 来获取它的一些详细内容。

函数声明如下：

```
napi_status napi_get_cb_info(napi_env env,
                             napi_callback_info cbinfo,
                             size_t* argc,
                             napi_value* argv,
                             napi_value* thisArg,
                             void** data)
```

- env: 上下文。
- cbinfo: 这次函数调用的信息，即一个 `napi_get_cb_info` 数据。
- argc: 用于获取本次函数调用的参数个数。
- argv: 用于获取本次函数调用的参数数组。
- thisArg: 用于获取本次函数调用的 this 对象。
- data: 用于接收我们在 `napi_create_function` 传入的 data。

5. 判断是否是构造函数调用——`napi_is_construct_call`

这个函数的作用类似于 3.7.9 节中提到的 `IsConstructCall()` 函数，用于判断该次调用是否为构造函数调用，即是否为使用 `new` 前缀进行调用。

函数声明如下：

```
napi_status napi_is_construct_call(napi_env env,
                                   napi_callback_info cbinfo,
                                   bool* result)
```

- `env`：上下文。
- `cbinfo`：这次函数调用的信息，即一个 `napi_get_cb_info` 数据。
- `result`：用于接收结果的参数，其为一个布尔值，代表是否。

6. JavaScript 的构造函数调用——`napi_new_instance`

这个函数用于通过类似于 `new` 的方式去调用一个传进来的构造函数，得到一个实例化的对象。函数声明如下：

```
napi_status napi_new_instance(napi_env env,
                              napi_value cons,
                              size_t argc,
                              napi_value* argv,
                              napi_value* result)
```

- `env`：上下文。
- `cons`：构造函数，一个 `function`。
- `argc`：调用构造函数所要传的参数数量。
- `argv`：调用构造函数所要传的参数数组。
- `result`：用于接收实例化出来的对象。

我们假设在 `global` 下有一个 `MyObject` 的构造函数如下：

```
function MyObject(param) {
  this.param = param;
}
```

那么在 JavaScript 中实例化它非常容易：

```
const arg = 'hello';
const value = new MyObject(arg);
```

如果使用 N-API 改写，它就会是这样的：

```
// 从 global 中拿到 `MyObject` 这个构造函数
napi_value global, constructor, arg, value;
napi_status status = napi_get_global(env, &global);
if (status != napi_ok) return;

status = napi_get_named_property(env, global, "MyObject", &constructor);
if (status != napi_ok) return;

// 定义参数为 `hello`
status = napi_create_string_utf8(env, "hello", -1, &arg);
if (status != napi_ok) return;

// 让参数变成一个参数数组，作用类似于
// napi_value argv[] = { arg }
napi_value* argv = &arg;
size_t argc = 1;

// 等同于 const value = new MyObject(arg)
status = napi_new_instance(env, constructor, argc, argv, &value);
```

7. 异步回调函数——`napi_make_callback`

`napi_call_function` 用于同步调用一个 JavaScript 的函数，而 `napi_make_callback` 与它相似。不过，`napi_make_callback` 通常是在一个异步操作后的流程中执行的（也就是说当前 JavaScript 的调用栈并没有上层堆栈信息的时候）。实际上 `napi_make_callback` 只是对于 `node::MakeCallback`¹ 的一个简单封装。

这个函数的声明如下：

```
napi_status napi_make_callback(napi_env env,
                              napi_value recv,
                              napi_value func,
                              int argc,
                              const napi_value* argv,
                              napi_value* result)
```

- `env`：上下文。

¹ 参照 5.2.4 节的 `Nan::MakeCallback()` 相关内容。

- `recv`: 调用函数时所需要使用的 `thisArg`。
- `func`: 需要调用的函数。
- `argc`: 调用函数时需要的参数数量。
- `argv`: 调用函数时的参数列表。
- `result`: 用于接收函数调用的返回值。

9.3.3 类的封装

我们回忆一下 Node.js 传统 C++ 扩展开发的方式，有一个 `ObjectWrap` 的类用于方便大家在 C++ 中封装 JavaScript 的类。N-API 同样提供了一系列的 API 来给大家定义 JavaScript 中的原型相关（即类）内容。

要在 N-API 中封装 JavaScript 类，大致分三步走：

- ① 通过 `napi_define_class` 来封装一个构造函数。
- ② 在 N-API 写的构造函数（实际上就是一个 `napi_callback` 函数）中，使用 `napi_wrap` 函数将一个我们需要包装的 C++ 层面的数据放进构造函数返回的内置字段中，这类似于 `ObjectWrap` 中的 `Wrap()` 函数。
- ③ 在需要使用内置字段的各函数中，通过 `napi_unwrap` 函数来获取内置字段，这类似于 `ObjectWrap` 中的 `Unwrap()` 函数。

1. `napi_define_class`

`napi_define_class` 用于定义一个 JavaScript 的类，并事先定义好一些信息：

- 一个构造函数，即一个 N-API 的 `napi_callback` 函数。
- 一些类的静态属性将以 `napi_static` 的形式定义。
- 一些原型链上的属性将通过非 `napi_static` 的形式定义。

它的原型如下：

```
napi_status napi_define_class(napi_env env,
                             const char* utf8name,
                             napi_callback constructor,
                             void* data,
                             size_t property_count,
                             const napi_property_descriptor* properties,
                             napi_value* result);
```

- `env`: 上下文。
- `utf8name`: 类名。
- `constructor`: N-API 形式的构造函数。
- `data`: 等同于 `napi_createfunction` 中的 `data`。
- `property_count`: 要定义的属性数量（无论是静态的，还是原型链上的）。
- `properties`: 要定义的属性数组，是一个 `napi_property_descriptor` 数组。如果其中某个属性需要为静态属性，则它的 `attributes` 值需要有 `napi_static`¹ 的属性，否则不需要有该属性。
- `result`: 用于接收这个类的 JavaScript 能使用的构造函数。

注意：我们通常会在 N-API 的某个函数中（或者在模块导出函数中）生成一个 JavaScript 的类，但是会保存下来后续再使用。在这种情况下，为了防止我们生成好的 JavaScript 类（实际上就是一个构造函数）被垃圾回收，我们通常需要使用 `napi_create_reference` 来创建一个该构造函数的持久引用计数，并确保在必要的时候其计数值始终大于 1。

2. `napi_wrap`

将一个 C++ 原生的数据类型（通常是我们自己定义的一个 C++ 层面类的实例）包装进一个 JavaScript 对象中。这一点在传统的 Node.js C++ 扩展开发中也有一样的操作，使用的是 V8 中的内置字段²。

具体的函数用法请参照官方文档（https://nodejs.org/docs/v8.4.0/api/n-api.html#n_api_napi_wrap）。

3. `napi_unwrap`

将一个包装进一个 JavaScript 对象中的 C++ 原生数据类型取出，它同样使用 V8 中的内置字段和相关的函数。

具体的函数用法请参照官方文档（https://nodejs.org/docs/v8.4.0/api/n-api.html#n_api_napi_unwrap）。

¹ 在 9.3.1 节中曾提到 `napi_property_attributes` 枚举的各种属性性质，其中 `napi_static` 的值是 `1 << 10`，即 1024。

² 可参照 3.7.5 节中内置字段的相关内容。

9.3.4 小结

本节简单介绍了使用 N-API 操作 JavaScript 的对象、函数、类的方式。由于未来充满了不确定性，因此本节内容较浅显，并没有做深入的解析，只是为了使大家对 N-API 有一个最初步的印象。

等到哪一天 N-API 真的成熟了，大家依然能通过阅读本节的内容来了解一些其开发的基本概念（虽然连基本概念也有变动），但是具体到各 API 的细节时，还是推荐大家直接阅读 Node.js 官方文档的 N-API 相关章节。

9.3.5 参考资料

- [1] N-API | Node.js v8.4.0 Documentation: <https://nodejs.org/docs/v8.4.0/api/n-api.html>.
- [2] ECMAScript® 2018 Language Specification: <https://tc39.github.io/ecma262/>.
- [3] ArrayBuffer - JavaScript | MDN: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer.
- [4] TypedArray - JavaScript | MDN: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/TypedArray.